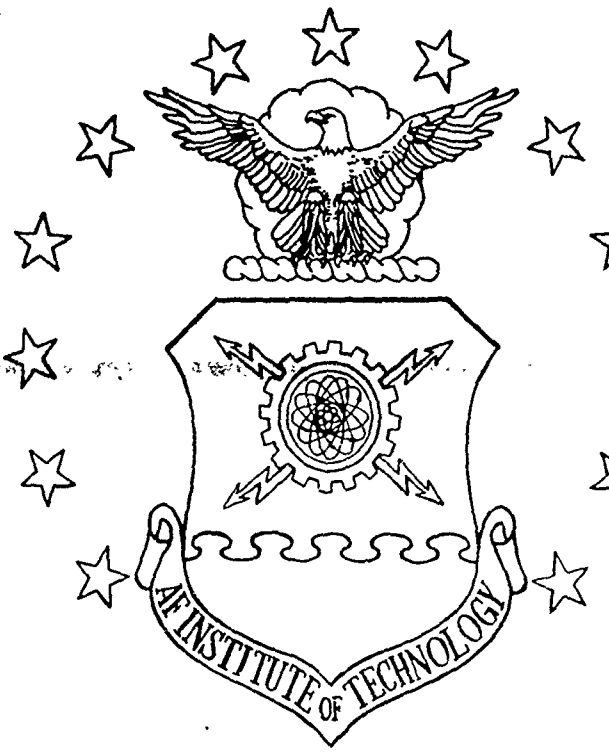


DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-10

An Animated Graphical Postprocessor for the Saber Wargame

THESIS

Gary Wayne Klabunde
Captain, USAF

AFIT/GCS/ENG/91D-10

AFIT/GCS/ENG/91D-10

An Animated Graphical Postprocessor for the Saber Wargame

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Gary Wayne Klabunde, B.A.
Captain, USAF

December, 1991

Preface

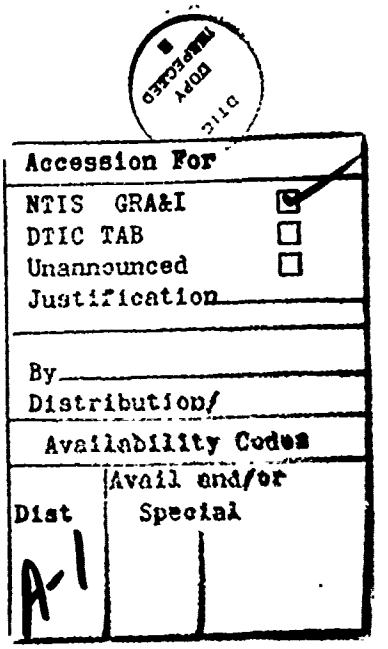
This thesis documents the design and implementation of an animated, graphical post-processor for the Saber wargame developed at the Air Force Institute of Technology. The Ada based post-processor utilizes the X Window System to provide the game players with the force status information necessary to plan and execute a theater-level air and land war. The report processor produces reports that can be viewed on the screen or printed in hardcopy form. The animation portion of the user interface allows the game players to see how the day's battle unfolded. They can see how their mission orders were carried out in addition to the enemy's response.

This effort was part of a team development. Other team members, developed the air and land simulation, system database, and input routines. The post-processor provides a stable framework for future interface enhancements and modifications.

I extend my gratitude to several people, without whose help, this thesis could not have been accomplished. First, I would like to thank my thesis advisor, Major Mark A. Roth. His experience, assistance, and guidance were of enormous help as I struggled with the technical and theoretical issues. I also wish to thank my committee members, Majors Michael Garrambone and Eric Christensen, for their hints, suggestions, and editing of the draft manuscripts. This thesis could not have been accomplished without the assistance of Captain Tim Halloran and others at the Air Force Wargaming Center. Their technical assistance in the intricacies of X and the Motif widget set were greatly appreciated. I especially want to thank my beautiful wife, Maria, who basically lived eighteen months as a single parent. I cannot repay her for the patience, understanding and support she provided during this research effort. Finally, I wish to thank my three year old son, Christopher, for making me smile when I needed it most. Daddy's coming home to play.

Gary Wayne Klabunde

Table of Contents



List of Figures

List of Tables

Abstract

One of the most cost effective ways to learn and hone the skills necessary to conduct and win a war is through the use of realistic computer simulations of conflict, or wargames. The Saber wargame was developed for just this purpose. Saber is a multi-sided, theater-level simulation developed by the Air Force Institute of Technology for the Air Force Wargaming Center. It models conventional, nuclear, and chemical warfare between aggregated air and ground forces. To aid in the realism, the effects of logistics, satellites, weather, and intelligence are represented. Saber provides an avenue for senior level joint service officers to improve their airpower employment skills.

This thesis documents the object-oriented design and implementation of the graphical post-processor for the Saber wargame. The user interface provides the game players with the force status information necessary to plan and execute a theater-level air war. The interface includes a report processor that produces reports for on screen viewing or printing. The system also provides animation capabilities to allow the game players to see how the day's battle unfolded in an effort to enhance the learning process.

The user interface was written in the Ada programming language using the X Window System and OSF/Motif widget set. Ada bindings developed by the Boeing Aerospace Corporation and the Science Applications International Corporation (SAIC) were used to interface to the various X libraries. These bindings were supplemented with bindings to a hexagon program written by the Air Force Wargaming Center.

The combination of the X Window System and the object-oriented philosophy proved effective in developing a user interface that is easy to use, predictable, and flexible. The system can be executed on any hardware platform that supports the X Window System. The use of the object-oriented paradigm should make it easy to enhance and maintain the interface.

An Animated Graphical Postprocessor
for the Saber Wargame

I. Introduction

> War to the hilt between communism and capitalism is inevitable. Today, of course, we are not strong enough to attack. Our time will come in fifty to sixty years. To win, we shall need the element of surprise. The western world will have to be put to sleep. So, we shall begin by launching the most spectacular peace movement on record. There shall be unheard of concessions. The capitalist countries, stupid and decadent will rejoice to cooperate to their own destruction. They will leap at another chance to be friends. As soon as their guard is down, we shall smash them with our clinched fist.
>
> – Dimitry Manuilski, Professor,
> Moscow's Lenin School of
> Political Warfare
> – 1930 –

At all times and all ways, the military forces of the United States must be prepared for war. The practicioners of war must learn proper techniques and master the tools at their disposal if they hope to be successful on the battlefield [13]. Short of war itself, the best way to learn and hone these skills is through military field exercises. Unfortunately, large, joint exercises require extensive planning and are extremely costly. A more cost effective approach to achieving the proper education or training is through the use of computer simulations of conflict, or wargames. This thesis presents the design and implementation of a graphical postprocessor and report generator for the Saber wargame.

1.1 Background

Saber is a multi-sided, theater-level wargame developed for the Air Force Wargaming Center. It models conventional, nuclear, and chemical warfare at the aggregated air and ground forces level with the effects of logistics, satellites, weather, and intelligence represented. Saber was designed to provide an avenue for improving the airpower employment

skills of senior Air Force leaders attending classes at the Air War College and the Air Command and Staff College. Specifically, Saber exposes senior level joint service officers to the types of high level decisions that must be made to plan and execute an air and land campaign at the theater level.

The ground portion of the exercise was modeled and developed by Captain Marlin Ness [36] in 1990 using the Ada programming language. Ness' object-oriented Land Battle program is generic and adaptable to any combat area in the world. It uses discrete events with fixed time steps to model conflict between land units at, or above, the division level.

In addition to the basic formulas for calculating attrition, Ness developed software for land units to carry out such missions as attack, defend, withdraw, and support. The units are located in and move across a series of interlocking hexagons. The hexagons provide a good way to represent the type of terrain being traversed, mobility impediments such as destroyed bridges or minefields, and the weather for a particular area. The output of Ness' model consisted of rather lengthy and somewhat cryptic reports.

Following Ness' efforts, Captain William Mann [31] laid the groundwork for the development of the air portion of Saber. Mann linked Air Force doctrine with a conceptual model's framework to design an air battle for a new air/land model called Saber. For any combat simulation to be credible, the foundations upon which the formulas are based must be known and understood. Mann's thesis effort, therefore, involved both developing the formulas and algorithms for stochastic air attrition as well as documenting the rationale and justification of the design decisions.

Mann altered the definition of land units and added theater air defense units, air bases, and aircraft packages. The missions the air units can fly include, among others, counterair, interdiction, and close air support. The aircraft packages can move through any of seven layers of air hexes corresponding to different altitudes above sea level. In size, a single air hex encloses seven of Ness' ground hexes.

In his thesis, Mann presented an overview of the input and output processes required for simulations as shown in Figure 1. He categorized output of a simulation into three

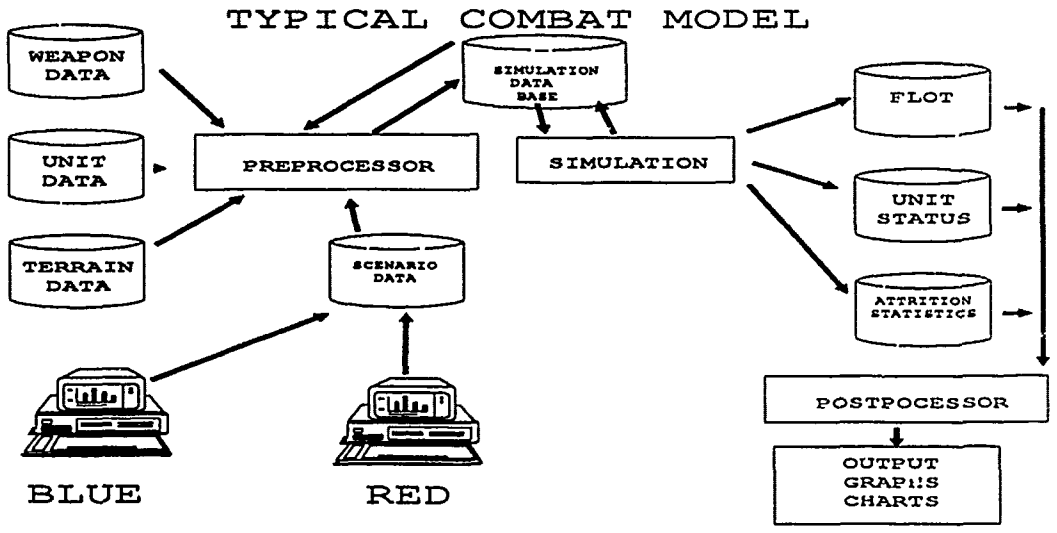


Figure 1. Mann's Typical Combat Model[31]

forms: raw data, processed reports, and image reports. Several examples of each type are given.

Captain Christine Sherry [5?] developed an object-oriented design of the air war using the framework provided by Mann. Sherry's Ada code implements the formulas and algorithms to determine battle outcomes. It was at this time that the air battle and Ness' land battle were integrated to form a single event driven simulation. This marriage was necessary so that aircraft strike missions and land units could cause attrition on each other.

The Saber wargame was designed to be executed at least once each day. Modelling conflict across these different executions requires that information about each unit or entity be saved in some manner. To accomplish this task, Captain Andre Horton [20] developed a relational database using the Oracle database management system. He designed numerous

tables to hold the state of the objects in a manner that reduces replication of data. Horton also developed the input screens that allow the players to enter their mission orders into the system.

1.2 Problem

One of the most important components of any wargame is the representation of output. The players of a wargame need some form of feedback in order to analyze the situation and make decisions concerning the future employment of their forces. For the most part, a graphical representation of the data enhances enjoyment and understanding on the part of the participants. Accordingly, this thesis effort was directed toward providing an animated, graphical postprocessor and report generator for the Saber wargame that provided the participants with force status information necessary to plan and execute a theater level air war. The graphical user interface was developed using the X Window System along with the Open Software Foundation's Motif [1] widget set.

1.3 Research Objectives

This thesis effort resulted in the development of a graphical postprocessor for the Saber wargame. In designing and implementing this postprocessor, the following objectives were set forth:

1. A graphical user interface will be developed to display battle outcomes as well as the position and status of forces. The interface should be easy enough for novice computer users to operate, while at the same time, it should not hinder the rapid progress of more experienced users.
2. The user interface will utilize animation to show movement of land units and aircraft packages. The animation will show the specific routes taken from the starting location to the destination. It will also show the locations at which attrition was experienced.

[1] OSF, OSF/Motif and Motif are trademarks of the Open Software Foundation, Inc.

3. Since the exact location of all enemy forces is never known, an intelligence filtering mechanism will be developed to provide estimated locations and strengths of suspected enemy units to opposing teams.

4. Weather plays an important role in the success of most combat missions. In an air war, it can affect the types of aircraft that can fly as well as the types of munitions they can carry [46]. Therefore, a system for reporting forecasted and actual weather will be developed.

5. While a picture may be worth a thousand words, the physical size of the computer screen limits the amount of information that can be displayed at any one time. Thus, it will also be necessary to provide output in a hardcopy form. Status information will be output in the form of reports, bar graphs, and charts.

1.4 Assumptions

The research and development efforts in this thesis were based on the following assumptions:

1. The code developed by Ness, Sherry, and Horton correctly creates and updates the databases.

2. The graphical interface is to be developed on a Sun 386i workstation. It is to execute on a Sun 386i or a Sparc Station II.

3. The graphical user interface is to be developed using X Windows and OSF/Motif.

4. The Ada programming language is to be used as much as possible.

1.5 Approach

The basic approach employed in this thesis effort consisted of the following steps:

1. Conducted research in the areas of graphical issues in wargaming, graphical user interfaces, and the X Window System. A proper understanding of these areas was essential to the accomplishment of this thesis effort.

2. Selected a design methodology appropriate for solving the given problem. No one methodology is correct for all projects. Some of the factors considered in making this decision included the subject matter, the product being produced, the desired features, and the associated risks.

3. Clarified the requirements using a paper prototyping system similar to that developed by Mark Kross [29]. In any programming project, an otherwise correct program is worthless if it does not meet the needs of the user.

4. Developed a preliminary object-oriented design. The design included the relationships among objects as well as their attributes and methods.

5. Iteratively accomplished risk assessment, detailed design, coding, and testing. This iterative process provided a way to control and measure changes being made to the system as it was being developed.

1.6 Standards

The importance of standards in any programming effort cannot be overlooked. When closely followed, standards can decrease the time it takes to become familiar with and truly understand a computer program. This can help with debugging a program during development as well as with maintenance efforts after delivery of the package. Thus, the following standards were adhered to during the development of the software:

1. Documentation shall be in accordance with the guidelines developed for AFIT by Dr. Thomas Hartrum [18].

2. The Ada *use* clause will only be used if absolutely necessary.

3. There will be a consistent use of naming and capitalization of variables and reserved words.

4. The behavior of the user interface will closely follow the *OSF/Motif Style Guide* [41].

1.7 Thesis Overview

Chapter II is a literature review of graphical issues that pertain to output in wargames, the qualities of good graphical user interfaces, and the X Window System. Chapter III briefly covers the design methodology chosen to successfully implement this project. Chapter IV then goes into the requirements for the display screens and reports. It also describes the high level and detailed designs for the system. Chapter V presents the design and implementation issues faced in this effort. And lastly, Chapter VI presents a summary of the project along with conclusions and recommendations.

II. Literature Review

2.1 Introduction

In order to begin developing a graphical user interface, we need to understand the user's requirements and the role the user interface is to serve in fulfilling those requirements. Once the "big picture" has been grasped, it is imperative that the user interface designer have some understanding of the factors involved in creating a satisfactory user interface. The last step in user interface development is to decide upon and master a computer language or system that can be used to effectively implement the design.

To achieve the required understanding, a review of current literature was conducted in the areas of war games, graphical user interfaces, and the X window system.

2.2 Wargames

Wargames are simulations that model conflict between opposing forces using complex rules to control entity movement and determine battle outcomes. In addition to attrition, they may also model logistics, command and control networks, intelligence methods, weather, troop morale, and military tactics. When used as an educational tool, their primary purpose is to assist the player in understanding the dynamics of warfare and the processes involved in combat. As Lt Col John Madden writes, they allow the participants to gain insights into decision processes that relate "...the principles of war, war fighting systems, and force employment decisions to military objectives of war" [23:11].

Peter Perla notes that a wargame must be interesting and relatively easy to play if it is going to make the players "...suspend their inherent disbelief, and so open their minds to an active learning process" [42:8]. Furthermore, the wargame must be accurate and realistic if the learning is to be meaningful instead of misleading. With this in mind, representation of warfare on a computer screen requires consideration of such factors as using map-based graphics, utilizing animation, displaying forecasted and actual weather, relaying intelligence information, and generating status reports. These factors are discussed in the following sections.

2.2.1 Map-Based Graphics. In today's world, wargames and graphics go hand-in-hand. Modern wargamers desire some way of visualizing the battle that is being simulated. However, graphics capabilities can also benefit the developer of a simulation. William Biles writes that are three basic ways in which graphics aid in simulations: [4:472]

- To enhance the simulation results
- To facilitate the debugging and production of simulation programs
- To provide an interactive dialogue with a running simulation

One of the more important uses of graphics in wargames involves representing a map of the battlefield on a computer screen. There are several options to consider when choosing the method of representation. These include:

- Two-dimensions versus three-dimensions
- Representation of elevation and contours
- Quality of the drawing (i.e., level of detail)
- Representation of terrain features
- Source of the drawing
- Coordinate representation (i.e., latitude and longitude, Universal Transverse Mercadian (UTM), Cartesian coordinates, and hexagon numbers)

Two of the most obvious sources of the maps include manually drawing the desired features or digitizing a map of the desired area. However, these methods are not as accurate or as fast in replicating the desired features as other methods available. [50]

Darryl Quick[44] utilized a database produced by the Central Intelligence Agency to represent a two-dimensional, low detail map of Europe for the Theater Warfare Exercise (TWX). The database, called Micro World Data Base II (MWDB-II), allowed for the representation of such graphical data as coast lines, country boundaries, lakes, and rivers. He used a conceptual series of layers to represent such features as geographic data, unit/base information, weather, and pop-up windows.

Quick reported a number of challenges which had to be overcome to use the MWDB-II. The first was developing a conversion routine to translate between the MWDB-II latitude/longitude system to the Cartesian coordinate system used in TWX. Secondly, the MWDB-II uses individual files to store particular types of information for the entire world. Quick found it more efficient in terms of speed to "...extract portions of each of the applicable files so that all the geographical information for a certain region would be stored together"[44:19]. A third problem faced was that the display routines provided with MWDB-II could not easily be integrated into other graphics software that displayed the additional layers of his conceptual model. Furthermore, the display routines did not lend themselves well to Quick's zoom and pan capabilities. The fourth challenge concerned accounting for distortion that occurs when mapping from a spherical surface to a two-dimensional map representation. However, Quick discovered that the amount of distortion introduced was insignificant for the European area he was using.

The MWDB-II is not the only source of precollected map data. In her thesis, Lieutenant Nora Stevens described the Joint Theater Level Simulation (JTLS) developed by the Jet Propulsion Laboratory. [55] This wargame uses map data provided by the Defense Mapping Agency. The graphics screen displays maps overlaid with text and standard military unit symbols. The Transportation Safeguards Effectiveness Model (TSEM) written by the BDM corporation uses still another source of geographical data. This model uses a U.S. Geological Survey Data tape produced from satellite and aircraft reconnaissance information. The data is formatted in the Universal Transverse Mercadian (UTM) mode and has elevation resolution accurate to 1 meter. BDM corporation used pre-existing programs to select a particular window of the world simply by specifying the degrees, minutes, and seconds of latitude and longitude.[50]

2.2.2 Animation. A computer wargame may or may not use animation as part of its graphical display. If properly used, animation can be beneficial to the military decision maker in that it provides a way to view the dynamics of the battlefield. The players can simply watch the battle unfold and see the outcomes of their decisions and what synergistic

effect their decisions had on the war.[5] Daniel Brunner lists the following benefits as being cited most frequently by advocates of animation [8:155]:

- Animation helps those who have created a simulation to 'sell' their quantitative conclusions to skeptical upper managers.
- Animation, used during the model development cycle, helps the model builder(s) build, verify, and validate the model.
- Animation, because it is flashy and fun, helps users and managers generate and maintain interest in exercising the analytical power of simulation, to the presumed benefit of everyone.

Given that animation has benefits to the wargaming community, the question remains of what methods are available to product animated graphics. Stephanie Cammarata presents a good summary of two animation techniques that have been commonly used in the past. She calls these the "display processor" approach and the "incremental graphics" approach. [9]

The display processor approach treats the graphical display as a process independent of the simulation. This process redisplays the entire picture at regular intervals. Some synchronization is required to ensure the simulation is not updating its state at the same time the display processor is creating a new image. The main disadvantage of this approach is that the entire simulation state must be redisplayed at each interval. This overhead can become costly if the update interval is too small or the simulation state is updated infrequently.

In the incremental graphics approach, the graphics display is updated as the simulation executes. Thus, the simulation controls the display by generating new images only when the state changes. Cammarata suggests that this method may improve the appearance of the display because only graphic attributes whose state has changed need to be redisplayed. [9] Cammarata claims that, unfortunately, neither approach just described lends itself well to efficient implementation in object-oriented simulations. As an alternative, she proposes a "graphics-delta" approach which combines the best features of each method. She writes, "The graphics-delta approach allows a user to declaratively specify simulation objects and attributes, and define corresponding graphical images which

support the simulation's graphic display" [9:509]. In this approach, a display processor determines what changes are necessary to produce a new image. The display is then updated as needed.

2.2.3 Weather. The impact of weather in war is well understood but sometimes overlooked. With the advent of all-weather aircraft, poor weather is playing a somewhat diminished role in affecting which aircraft can fly. However, there are still a large number of aircraft in the world that cannot fly in bad weather. Furthermore, weather plays an important part in the success of the ground mission. As was recently witnessed in Saudi Arabia, sudden sand storms can bring all ground activity to a virtual standstill. Weather also affects the success of reconnaissance missions. If fog or clouds obscure the ground, enemy activities may go undetected. James Dunnigan describes the effects of weather as follows:

> Rain, snow and excessive humidity cut mobility and the efficiency of weapons and troops. Fog, clouds and mist obstruct observation. Fog aids the attacker by masking his troops from enemy weapons....Extremes in hot and cold temperatures have adverse effects on troops and machines, as will high winds. [13:484]

In a wargame, the same weather forecast may be applied to the entire theater of operations. Alternately, the theater may be divided into zones, with each zone having one type of weather. Marlin Ness suggested six types of weather for the land battle portion of Saber. Each hex was assigned a value ranging from excellent down to very poor. [36] William Mann, on the other hand, suggests only good, fair, and poor weather be represented [31]. The type of weather predicted for an area may affect the weapons loads carried by the strike aircraft. The three types of weather suggested by Mann are sufficient to model this aspect of the wargame [46].

2.2.4 Intelligence. The primary function of intelligence in war is to gain timely and accurate information about the enemy while keeping him from doing the same. However, Dunnigan notes that due to limited reconnaissance resources and rapidly changing conditions during war, you can only reveal about 10 to 20 percent of the enemy's actual activities each day [13].

There is a wide spectrum of activities involved with intelligence gathering, assimilation, and dissemination. However, an explanation of the full gamut of these activities is beyond the scope of this research. For the purposes of the Saber wargame, the majority of intelligence data comes from reconnaissance satellites and aircraft, returning strike mission aircraft, and ground units in contact with enemy forces. When simulating intelligence gathering and reporting in a wargame, Perla writes that it is common practice to limit the knowledge reported to the players in ways consistent with the player's actual capabilities to obtain the data. He further suggests that to mimic "the fog of war," game designers or controllers should restrict access to certain information and deliberately introduce inaccuracies in the data that is reported. [42]

Mann and Ness discuss a way of modeling intelligence through the use of intelligence indices and filters [31, 36]. Each unit has an intelligence index which is raised when the entity has been observed via reconnaissance or contact with enemy forces. Ness also suggests that the index value be decreased over time if the unit has not been recently observed. The intelligence index is used to calculate an intelligence filter for the unit. A random number generator is then used to determine the actual amount or type of information to be provided to the players.

2.2.5 Report Generation. In an educational wargame simulation, it is desirable to provide some type of feedback to the participants. This feedback assists the player in understanding the impact of his force employment decisions and provides the necessary information for him or her to make future decisions. Perla describes the qualities of the feedback as follows:

> The information provided to the player should be organized in a way that gives him a sense of the possible effects of the important factors, along with enough extraneous details to make the task of sorting out precisely what is important sufficiently difficult to be realistically challenging and educational. [42:199]

However, too much information can have adverse effects. If the players are overwhelmed with data, they may become frustrated and lose interest in the game. It is, therefore, important that the players can find the crucial data among the superfluous information. Robert Sheridan writes that:

> Even the most sophisticated and elegant simulation is worthless if the results of the simulation cannot be interpreted. Large engagements involving many players and complex interactions producing reams of listings may lend themselves to misinterpretations and negative effects. [50:822]

Mann takes the position that an overabundance of data is preferable in order the keep the model flexible [31]. He believes the importance of a piece of data depends on the perceptions of the individual players. Therefore, he suggests that the data should be presented in both raw and processed forms. Raw reports consist of such things as data input echo reports, detailed logistics reports, and the output of a transaction file. Processed reports on the other hand, contain the same information as the raw reports, but in a summarized and aggregated form. Of the processed reports, the ones most valuable to high-level decision makers are often those which portray rates of change [3].

The information provided to the players commonly takes the form of tables or charts. Tables present information in a row and column format in which some form of human interpretation is still required to search for relevant trends and patterns. Charts (e.g., pie charts or bar charts) portray information through graphics that can be more quickly understood by the players [31]. Glenn Simon describes a line graph used in the Small-Unit Amphibious Operation Combat Model in which connected data points representing such things as attrition are plotted once for each time interval [52]. The line graph shows the correlation between the measured quantity and the passage of time.

A map-based graphical representation of the battlefield can also be used to display information about the current situation. It is much easier to scan a map to determine location of forces and the terrain in which they are located than it is to do the same thing by looking at a table or chart. Some models currently being developed at the Air Force Wargaming Center use map-based graphics with units represented using standard military symbols and the unit's name under the symbol. Mann suggested new wargames should follow in the same manner, but each unit should be augmented with "decision graphics" [31]. U.S. Army FM 101-5-1 describes decision graphics as two small circles divided into thirds or quarters. One circle represents the degree of mission accomplishment, while the other can be used to represent the status of particular items of interest for a unit. The

amount that the circle is filled in or the color with which it is filled signifies different things. [12] For example, a completely darkened circle means that a unit is unable to perform its assigned mission.

2.3 Graphical User Interfaces

The user interface is the component of the application through which the user's actions are translated into one or more requests for services of the applications, and that provides feedback concerning the outcome of the requested actions [35]. The design of efficient and easy to use user interfaces is receiving increased attention these days. Most people now realize that if an application has a user interface that is "unfriendly" or difficult to use, it is probably going to sit on the shelf unused.

2.3.1 User Interface Design. While much has been written recently c, the subject of user interface design, it is hard to define exactly what is meant by a "good" user inte ce. Often, the closest one can come to a definition is an enumeration of qualities a user interface should have. Accordingly, it is not easy to design a user interface. Brad Myers describes user interface design as more of an art than a science. However, he does list some things to consider when producing a design [33]:

- *Learn the application.* In order to determine what data to display and how best to display it, the designer must have a good understanding of the functionality of the system. This is often one of the most significant steps in interface design as a poor understanding can be difficult to overcome once the design progresses.
- *Learn the users.* The designer must determine the skill levels of the intended users, their backgrounds, and the amount of training likely to be needed.
- *Learn the hardware and environmental constraints.* Is the system going to be run on a particular type of machine? Will special input or output devices, such as mice or plotters, be used?
- *Evaluate similar products.* The designer should study the user interfaces of similar systems and of systems in the same environment.

- *Determine the support tools.* There are many toolkits available to assist in the design and implementation of user interfaces. Also, user interface management systems (UIMS) are becoming more popular as a means of increasing productivity in the user interface design.
- *Plan to incorporate Undo, Cancel, and Help from the beginning.* It is very difficult to try to add these functions after the system is under development. The nature of the actions impacts the design of the application's data structures.
- *Separate the user interface from the application.* The user interface and the application should be modularized with the design of the former being based on the functionality of the latter.
- *Design for change.* The user interface will change more than the functionality of the application. These changes frequently will be based on customer reaction to the delivered system.

Two of the items in the above list deserve a broader discussion. These are the support tools and the separation of the application from the interface. As previously mentioned, the two major types of tool for user interface design are toolkits and user interface management systems (UIMS). One problem with toolkits is that it is often difficult to determine what part of the toolkit to use to perform a particular function. Furthermore, since the work must be done over and over with each new application, consistency between systems is in jeopardy [28]. UIMS, on the other hand, are designed to aid in "rapid development, tailoring and management of the interaction in an application domain across varying devices, interaction techniques and user interface styles" [30:33]. This may include such things as handling user errors, providing helps and prompts, and validating users inputs.

Separating the user interface software from the application software has many attractive benefits. Typical user interface design consists of one or more prototypes offered to the user for review. The user then evaluates the interface and offers suggestions for improvement. If the application and the user are closely interwoven the user interface designer may have difficulty making the suggested improvements. The job can be much easier, however,

if the functionality of the application is separated from the user interface. Pedro Szekely lists the following benefits of minimizing dependencies between the application and the interface [56:45]:

- The user interface can be packaged into components that can be reused in other interfaces.
- The user interface can be changed without impacting the functionality.
- Multiple user interfaces can be developed for a single application, each one tailored to a different class of users, or to a different set of input and output devices.
- The functionality of an application can be called from another program directly, without simulating the input required by the user interface.
- The user interface can be specified by means other than programming, for example, by interactively drawing and demonstrating how the interface should behave.

2.3.2 User Interface Characteristics. Whatever design method is used, effective user interfaces frequently have certain qualities. Brad Myers lists the following attributes of so-called "good" user interfaces [34]:

- *Invisibility:* The user interface should be transparent to the user, such that the user has the sense that he is directly manipulating "real" objects on the screen. The user interface should not interfere with the operator's concentration on the task being performed.
- *Minimal training requirements:* No more than an hour of training should be necessary before the user can be productive on the system.
- *High transfer of training:* The system's appearance and performance should be similar to other systems dealing with the same subject matter. This external consistency between systems will help reduce training times when switching from system to system.
- *Predictability:* The objects and operations should perform similarly across contexts of the system. This internal consistency leads to a system where users can anticipate how the computer will behave.

- *Easy to recover from errors:* The designer should assume users will spend around 30% of their time in error situations. Thus, the user interface should allow the operator to easily recover from these errors. In addition to providing a "back-up" capability, the system should inform the user of the cause of the error and how to avoid repeating the same mistake.
- *Experts operate efficiently:* It is generally agreed that most programs should be oriented towards the novice or intermediate user who cannot consistently use the system without some form of assistance [19, 58]. However, the system should also provide the capability for experienced users to work quickly, unencumbered by an overly simplistic interface.
- *It is flexible:* The user interface should allow users to operate in the manner with which they're most comfortable. Users should be able to customize certain attributes to their own style and taste.

Consistency in user interface design is generally regarded as a fundamental requirement [37, 49, 53, 54]. User interfaces should be consistent both within themselves and with similar products used in an organization. Consistent user interfaces:

- enhance the transfer of skill across systems.
- allow users to predict system performance.
- allow users to focus on changes in the presented data.

Jonathan Grudin, however, feels the emphasis on consistency is misdirected. He feels that "when user interface consistency becomes our primary concern, our attention is directed away from its proper focus: users and their work" [16:1164]. Furthermore, he argues that interface consistency is an unattainable goal. Grudin gives several examples to illustrate his point. One of these deals with the selection of default menu choices. One approach for selecting the default would be to highlight the item the user is most likely to select next. For example, after a user performs a "Cut" operation, the default for the next menu selection should probably be "Paste". However, certain functions, when executed, may be irreversible. For these functions, a confirmation screen is often displayed to make

sure the selected action really is desired and to give the user an opportunity to change his mind. Experienced users will want to go ahead and perform the selected action the majority of the time. However, it is safer in these cases to set the default to the "Cancel" selection – the selection the user probably doesn't want. But, this violates the consistency rule whereby the item most likely to be selected is highlighted as the default. Thus, Grudin feels consistency should be considered in user interface design, but it is more important to have a good understanding of the users, their task, and their environment.

2.3.3 User Interface Styles. Each user interface has a certain style or method of getting information from the user. A user interface may have only one style or it may have several, where different styles are used at different times or for different reasons. The designer's choice of style will have a big impact on the software design. Some of the more common styles are question and answer, command language, windows-icons-menus-pointers (WIMP), forms, dialogue boxes, and direct manipulation.

Of the above listed styles, WIMP interfaces are among the most popular with novice and intermediate users. It is relatively easy to learn and use this style of interface. Users no longer must remember command syntax. Instead, command entry is accomplished by pointing to a menu item. An important benefit is that *recall* of commands from memory has been replaced by the easier *recognition* of functions [10]. WIMP interfaces also benefit from the capability of displaying information in multiple windows on the screen at the same time.

However, as Ian Sommerville points out, menu based systems suffer from a number of problems [54:266]:

- Certain classes of action, particularly those queries which involve logical connectives (and/or/not) are awkward or even impossible to express using a menu system.
- If there are a large number of possible choices, the menu system must be structured in some way so that the user is not presented with a ridiculously large menu. The most common structuring technique is hierarchical.
- For experienced users, menu systems are sometimes slower to use than a command language.

To accommodate experienced users, menu systems will often also accept some type of abbreviated command language, so that many of the menu selections can be bypassed. Another problem with menu systems is that users may find themselves lost in the myriad of levels of menus. One way around this problem is to show, in a small window, a graphical representation of the menu hierarchy and where the user currently is in the hierarchy. Alternatively, the user interface may retain all menus and submenus on the screen until the user selects an executable command. This allows the user to review the selections made up to a given point in time.

The actual selection of a menu item can be done in a couple of ways. One method is for the selected action to be performed immediately when the user indicates his selection. This method may be appropriate when speed is of more importance than accuracy. However, Smith and Mosier recommend a "dual activation" method for menu item selection [53]. With this method, menu selection is accomplished in two steps. The first step involves designating the selected option by movement of the cursor. The second step involves a separate action to cause the selected menu item to be processed. This step may be implemented by having the user select a separate box labeled "ENTER". Apple Computer calls this same method the "noun-verb" principle [2].

2.3.4 User Guidance. User guidance refers to system documentation, the on-line help system, and messages sent as a result of user actions. This is an area that doesn't always receive the attention it deserves. However, it should be considered at every stage of interface design because of the significant contributions it can make to effective system operation [53]. According to Smith and Mosier,

> The fundamental objectives of user guidance are to promote efficient system use (i.e., quick and accurate use of full capabilities), with minimal time required to learn system use, and with flexibility for supporting users of different skill levels. [53:291]

Often, the first impression a user gets of a system is from error messages [54]. Thus, the interface designer should make an effort to write error messages that are both polite and constructive without being offensive. When possible, the error message should suggest

how the user might recover from the error. Also, the user should have the option of getting a help message to give insight as to the cause of the error.

It is difficult for an interface designer to anticipate the level of help users will need. To accommodate all types of users, the help system should provide different levels of help. When the user first requests help, the system should provide a brief overview of the topic and give the user the capability to request a continuation of the help. Each successive level of help would give greater detail on the subject [58].

2.4 The X Window System

User interfaces using some type of windowing system are fast becoming a common feature of most computer systems. As a result, users tend to expect all application programs to have a professional, polished user-friendly interface.[62] The X Window System provides the mechanism to achieve this goal as well as many others described in the previous section.

The X Window System, or X, is a device independent, network transparent windowing system that allows for the development of portable graphical user interfaces [43, 48, 63]. It was developed in the mid 1980's at the Massachusetts Institute of Technology (MIT) in response to a need to execute graphical software on several different types of incompatible workstations. Robert Scheifler of MIT and James Gettys of Digital Equipment Corporation (DEC) developed X with the primary goals of portability and extensibility [48]. Another major consideration was to restrict the applications developer as little as possible. As a result, X "...provides mechanism rather than policy" [24:xvii].

To achieve these goals, the X Window System relies on the fundamental principles of network transparency and a request/event system. Software toolkits are then layered on top of the basic system to provide an easier programming environment.

2.4.1 X Window System Principles

2.4.1.1 Network Transparency. Oliver Jones describes network transparency as the capability for X application programs running on one CPU to show their output

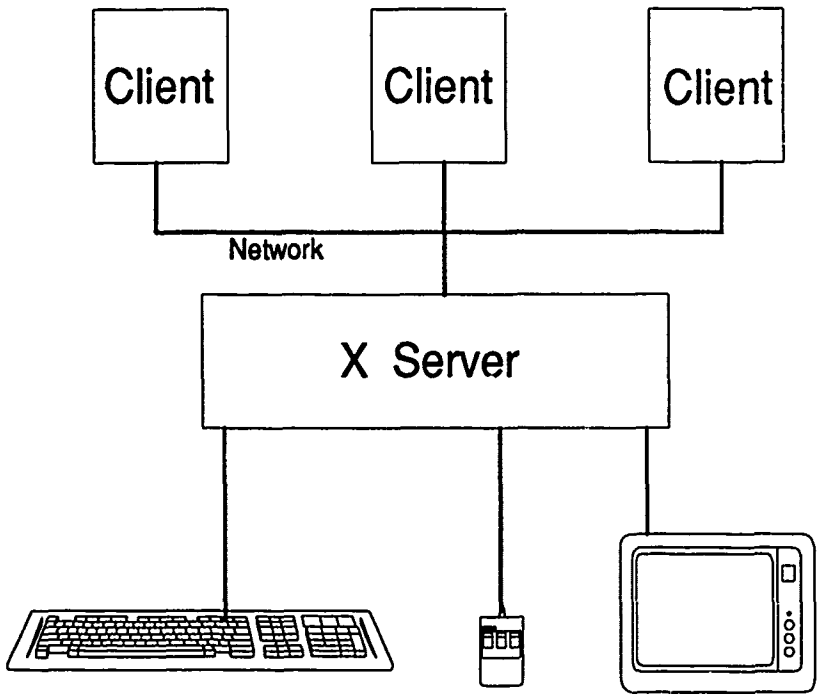


Figure 2. The X Client-Server Model

and receive their input "...using a display connected to either the same cpu, or some other cpu" [27:4]. The X Window System achieves this transparency using a client-server model.

In X, each workstation that is to display graphical information (i.e., windows or their contents) must have a process called the X server. According to Douglas Young, the X server "...creates and manipulates windows on the screen, produces test and graphics, and handles input devices such as a keyboard and mouse" [63:2]. A client, on the other hand, is any application program that uses the services of the X server.

Clients and servers use the X protocol to communicate with each other over a network. As Figure 2 shows, many clients can connect to a single server. Although not shown, a client can also be simultaneously connected to several X servers. In X, the client(s) and server can reside on the same physical machine, or they may be on separate machines.

2.4.1.2 Requests and Events. The network protocol mentioned in the last section is the method with which clients and servers communicate. This section discusses the mechanisms used to carry out the communication. The clients and servers communicate with each other by sending requests and event notifications, respectively.

When a client wants to perform some action on the display, it communicates this desire by issuing a request to the appropriate X server. Young states:

> Clients typically request the server to create, destroy, or reconfigure windows, or to display text or graphics in a window. Clients can also request information about the current state of windows or other resources. [63:4]

The X server, conversely, communicates with clients by issuing event notifications. Event notifications are sent in response to such user actions as moving a mouse into a window, by pressing a mouse button, or pressing a key on the keyboard. The X server also sends event notifications when the state of a window changes [63]. Applications programs act on these events by registering callbacks with the X Window System. A callback is simply a procedure or function that is to be executed when a specific event occurs.

Because of the reliability of the network, events and requests are sent asynchronously and data can be sent in both directions simultaneously [47]. This configuration makes for faster communication since the clients can send requests at any time and need not wait for an acknowledgement. The protocol guarantees the messages will be received in the proper order. Furthermore, there is no need for clients to continuously poll the server for information. "Instead, clients use requests to register interest in various events, and the server sends event notifications asynchronously" [47:xviii].

2.4.1.3 Basic Components. The X Window System was designed to provide the mechanisms for the application program to control what is seen on the display screen. The programmer is not constrained by any particular policy. These mechanisms are embodied in a library of C functions known as Xlib. The Xlib routines allow for client control over the display, windows, and input devices. Additionally, the functions provide the capability for clients to design such things as menus, scroll bars, and dialogue boxes.

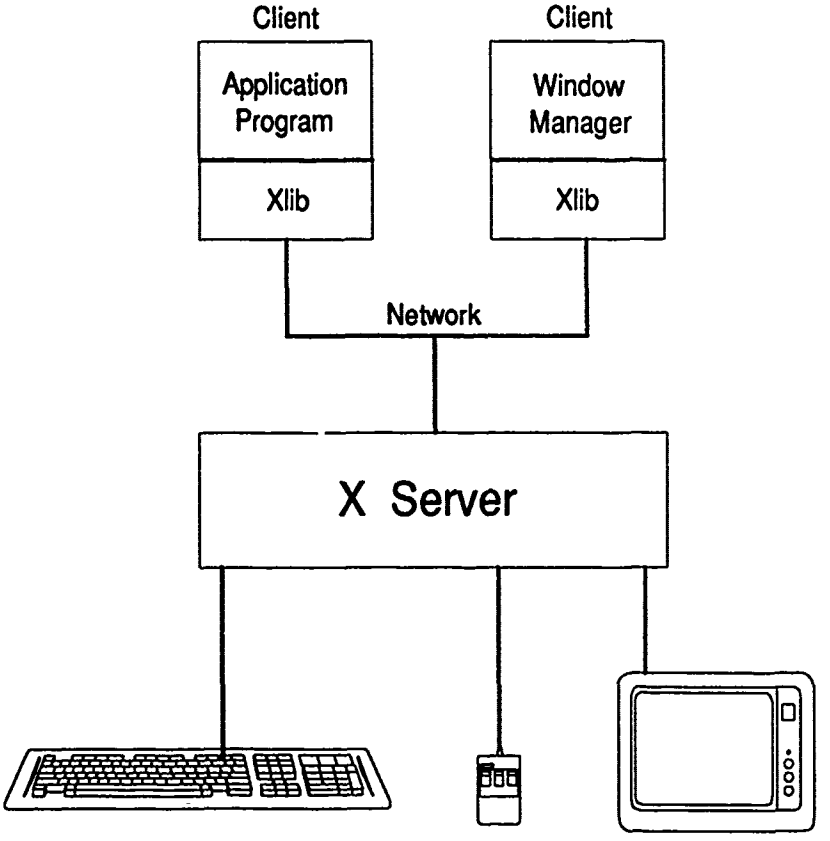


Figure 3. Basic X Environment

Most X application programs make use of a special client program called a window manager. This program utilizes the mechanisms of Xlib to relieve the application program of such tasks as moving or resizing windows [47]. Brad Myers writes that a window manager helps the user monitor and control different activities by physically separating them into windows on the computer screen [32].

Figure 3 represents the most basic X environment. In this diagram, an application program and a window manager operate as separate clients connected to a single server.

2.4.2 Toolkits. While applications programmers can use the Xlib routines to accomplish any task in X, many find the low-level routines tedious and difficult to use. Jay Tevis[57] noted that the simple action of creating and customizing a new window on the display takes at least 24 calls to Xlib. To simplify the development of applications pro-

grams, many toolkits have been developed. Toolkits can be viewed as libraries of graphical programs layered on top of Xlib. They were designed to hide the details of Xlib, making it easier to develop X applications.

There are several toolkits available today. Some of the better known ones include: the X Toolkit (Xt) from MIT, the Xrlib Toolkit (Xr) from Hewlett-Packard (HP), Open Look and XView from Sun Microsystems, and Andrew from Carnegie Mellon University. Of those listed, Xt is one of the most popular [22]. Along with Xlib, it is delivered as a standard part of the X Window System.

Xt is an object-oriented toolkit used to build the higher level components that make up the user interface [22]. It consists of a layer called the Xt Intrinsics along with a collection of user interface components called widgets. Widget sets typically consist of objects such as scroll bars, title bars, menus, dialogue boxes and buttons. In keeping with the X philosophy, the Xt Intrinsics layer remains policy free. As such, it only provides mechanisms that do not affect the "look and feel" (outward appearance and behavior) of the user interface [62]. These mechanisms allow for the creation and management of reusable widgets. It is this extensibility along with its object-oriented design that makes the X Toolkit attractive to user interface designers [60].

It is the programmer's choice of a widget set that determines the high-level "look and feel" of the user interface. Just as there is no "standard" toolkit, there are many different widget sets supported by Xt Intrinsics. However, as Young writes, "...from an application programmer's viewpoint, most widget sets provide similar capabilities" [63:12]. Some of the more popular widget sets include the Athena widget set from MIT, the X Widget set from HP, and the Motif widgets from the Open Software Foundation.

The Open Software Foundation (OSF) was formed in 1988 by a group of UNIX vendors including, among others, IBM, HP, and Sun Microsystems. The Motif widget set they created is designed to run on such platforms as DEC, HP, IBM, Sun, and Intel 80386 based architectures [25]. Eric Johnson list three advantages to using Motif [25:4]:

1. Motif provides a standard interface with a consistent look and feel. Your users will have less work to do in learning other Motif applications, since

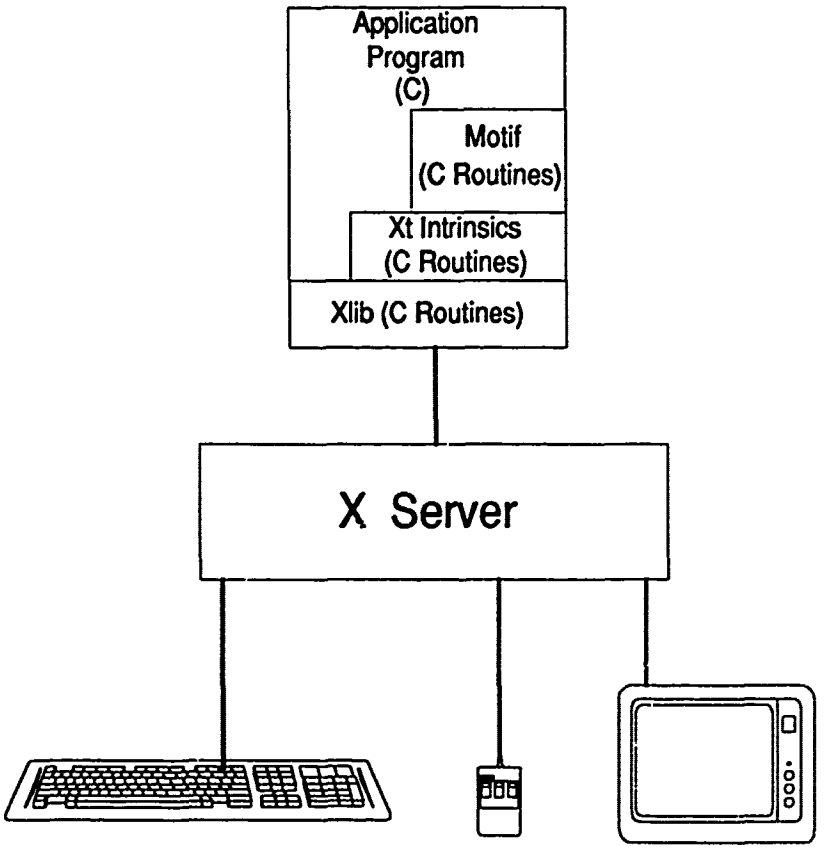


Figure 4. Typical X Windows Configuration

much of the work learning other Motif applications will translate directly to your applications.

2. Motif provides a very high-level object-oriented library. You can generate extremely complex graphical programs with a very small amount of code.
3. Motif has been adopted by many of the major players in the computer industry. Many of your customers are probably using Motif right now. You'll do a better job selling to them if your applications are also based on Motif.

Structurally, the Xt Intrinsics is built on top of Xlib. The Motif widget set, in turn, relies on the functions provided by the Xt Intrinsics. A typical application program may make calls to the widget set, the Xt Intrinsics, or Xlib itself during its execution. This configuration is illustrated in Figure 4.

Many user interface designers elect to design their own widget sets. Some do it for the challenge. Others design their own widgets out of necessity. A user interface designer may have a need for a special widget not provided by any available widget sets. However, designing custom widgets decreases the portability of the user interface code and of the application code in general [22].

2.4.3 Ada and X Windows. Originally, Xlib, Xt Intrinsics and most widget sets were written in the programming language C. Until a few years ago, there was no way for an application program written in Ada to use the X Window System. Recent efforts have taken two approaches: Ada bindings to X and Ada implementations of the X libraries.

2.4.3.1 Ada Bindings to X. In 1987, the Science Applications International Corporation (SAIC) developed Ada bindings to the Xlib C routines. Their work was performed under a Software Technology for Adaptable Reliable System (STARS) Foundation contract, and is therefore in the public domain. According to Kurt Wallnau, "...a substantial effort was made to map the C data types to Ada, and do as much Xlib processing in Ada as possible before sending the actual request to the C implementation" [60:5]. The actual Ada interface is accomplished through the use of Ada *pragma interface* statements [21]. Put simply, the *pragma interface* construct allows an Ada program to call subprograms written in another language [11]. Figure 5 shows the configuration of an Ada program using the SAIC bindings to interface with Xlib. In this figure, the application program has no access to any toolkits or widget sets.

Jay Tevis[57] successfully used the SAIC bindings in the development of a user interface for a CASE tool. However, he found the excessive number of required SAIC function calls to be a hindrance to effective program development. To offset this situation, he developed a Machine-Independent Ada Graphical Support Environment (MAGSE). MAGSE is basically an Ada toolkit/widget set similar to, but much smaller than, the X Toolkit.

In a manner similar to that used by SAIC, the Boeing Corporation recently developed Ada bindings to a large subset of the Xt Intrinsics and the Motif widget set. Their code also provides access to a very limited subset of Xlib functions and data types. Like the

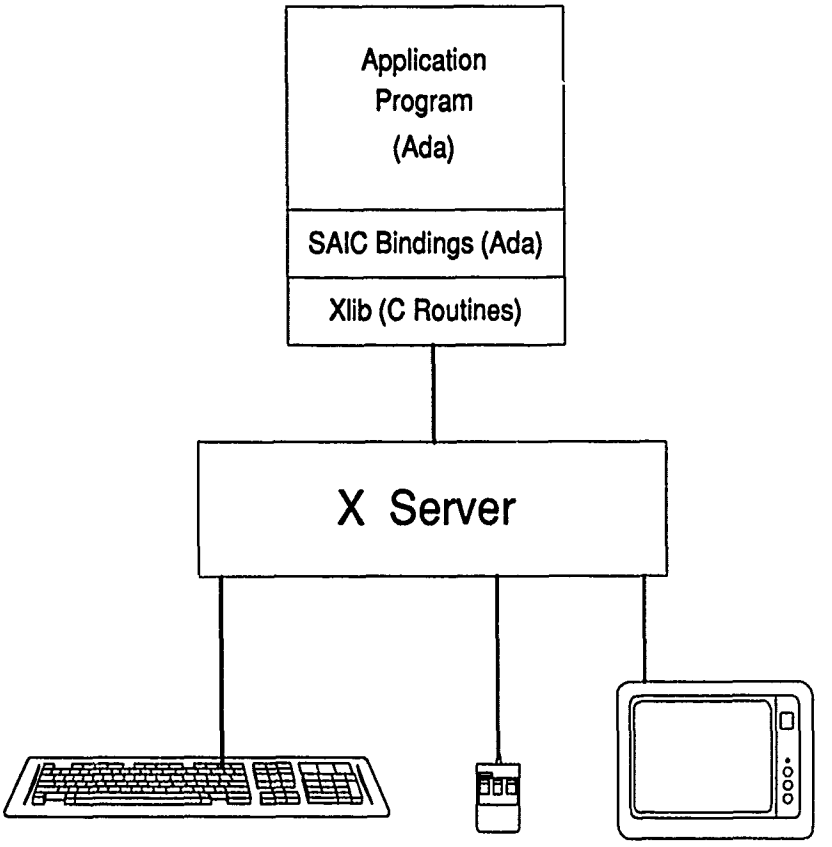


Figure 5. Application Program Configuration Using the SAIC Bindings

SAIC code, Boeing's effort was sponsored by a STARS contract[26]. For the most part, the subroutine names and parameter lists closely mirror the actual C routines. Also, Boeing added a few subprograms to assist in the building of some commonly used parameter lists. The bindings require the Verdix Ada Development System (VADS) version 5.5 or higher to execute. While the documentation on the software is relatively sparse, it does indicate which modules would require changes in order to port the bindings to other systems.

Figure 6 shows the configuration of an Ada program using only the Boeing bindings. The dashed lines indicate that a small portion of the Xt Intrinsics and Motif functions are unavailable to the Ada program. Also, the application program cannot access the majority of the Xlib functions.

The Ada application program accesses the Xt Intrinsics and Motif routines by calling

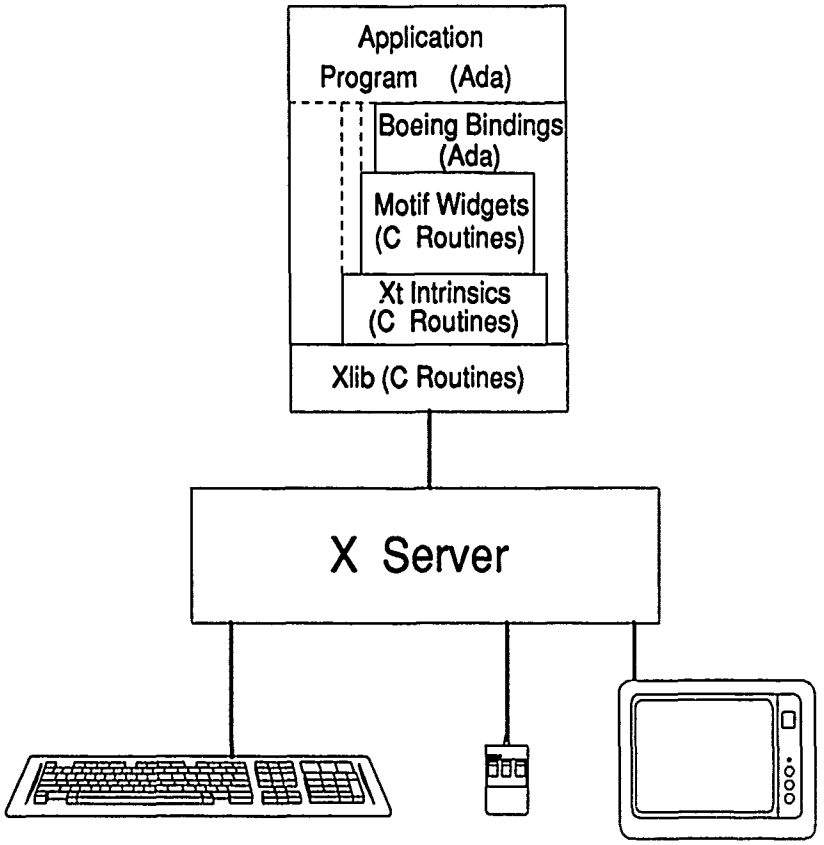


Figure 6. Application Program Configuration Using Boeing's Bindings

the appropriate subprogram in the bindings. For the most part, the bodies of the called subprograms contain code to convert the Boeing data structures and types to the types needed by the corresponding C code. The subprogram bodies then call internal procedures or functions that are bound to the Xt Intrinsics or Motif routines passing in the converted parameters.

The bindings developed by Boeing and the SAIC are available at no additional cost to the Department of Defense. Recently, several other corporations have also developed bindings that are available for purchase [1]. These companies have basically taken one of two approaches. Some have followed the approach taken by the SAIC and Boeing. Others, such as Hewlett-Packard, took an alternative approach. To alleviate the need for much of the type conversion used by the SAIC and Boeing bindings, Hewlett-Packard binds the

Ada subroutines directly to the corresponding C code. This results in very little code in the package bodies. To accomplish this, they make heavy use of Ada access types.

2.4.3.2 Ada Implementations. The USAF Electronic Systems Division recognized the need to write X Windows application programs in Ada at a higher level than through Xlib alone. In 1989, they sponsored a STARS Foundation contract to further research the capabilities of interfacing Ada and the X Window System [22]. The resulting reports documented efforts at integrating Ada with the X Toolkit (Xt).

As part of this STARS contract, Unisys Corporation developed an Ada implementation of (not bindings to) the X11R3 version of the Xt Intrinsics. "Ada/Xt," as it is called, "provides an intrinsics package which provides the functionality of Xt used to manage X resources, events and hierarchical widget construction" [61:1]. This software package uses a modified and corrected version of the SAIC bindings to interface to Xlib. Ada/Xt also includes a sample widget set consisting of ten Athena widgets and two HP widgets [61].

Unisys elected to develop an Xt implementation rather than Ada bindings, as SAIC did. The reasons for this included [60:9-10]:

1. The issue of widget extensibility. Ada bindings would require that new widgets be programmed in C.
2. The issues of inter-language runtime cooperation.
3. The issues of runtime environment interaction.

Figure 7 represents a typical Ada application program using the Ada/Xt interface. The Ada application code can make use of the provided widgets, make calls to Ada/Xt, or make calls directly to the Xlib via the modified SAIC bindings. Thus, the full flexibility of an X application program written in C is maintained.

2.5 Summary

This chapter consisted of a literature review in the areas of wargames, graphical user interfaces and the X Window System. The section on wargames concentrated on those issues important to designing and implementing a graphical display of a simulated

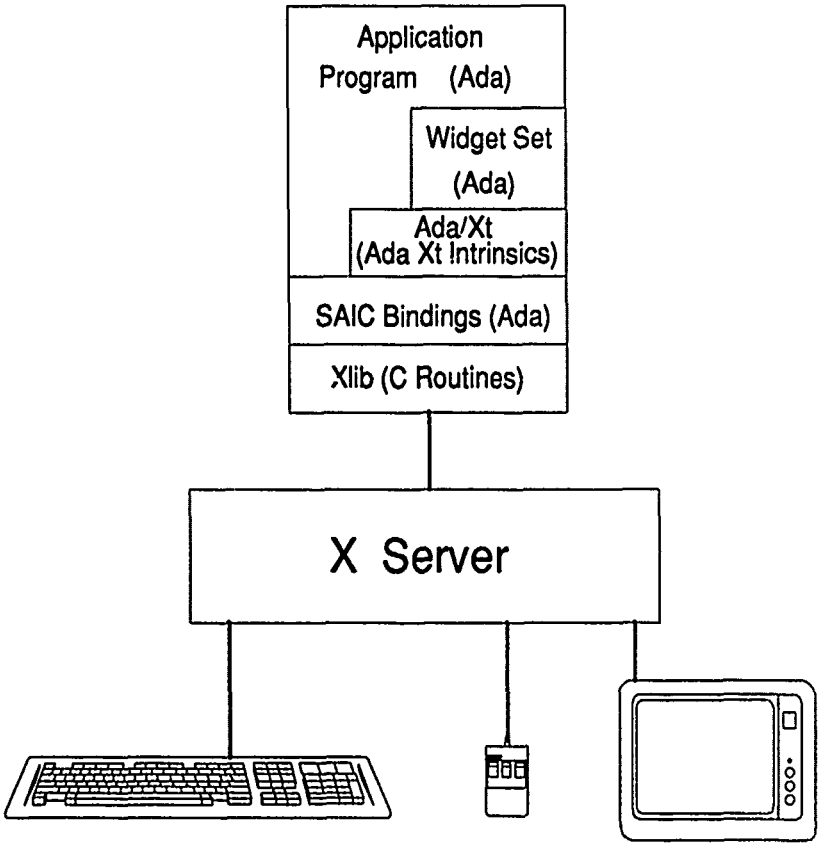


Figure 7. Application Program Configuration Using Unisys' Ada/Xt

battlefield. Basically, the section discussed "what" should be presented to the user and the benefits of using animation in the presentation. It illuminated the important categories of information such as weather, intelligence and statistical reports that should be provided to the game player to assist in the decision making processes. The review of graphical user interfaces identified some of the desirable qualities of user interfaces. This section discussed "how" a user interface should be designed so that the users will feel comfortable with the system and can be more productive. Lastly, an overview of the X Window System and its extensions was presented. The X Window System is a tool user interface designers can use to construct professional, and hopefully, user-friendly interfaces. This section culminated with a look at how application programs written in the Ada programming language can interface to the X Window System. The next chapter presents an overview of some popular methodologies used to develop software.

III. Design Methodologies

An important part of any software development project is the overall model or methodology for accomplishing the task. The methodology outlines the steps to be taken from inception through implementation to retirement. It provides an organized approach to software development and allows for management of the development effort.

3.1 Current Methodologies

Currently, there is no one standardized methodology being practiced. Unless a method has the capability for flexibility, it is doubtful that any particular one will be perfect for all software development. However, some organizations have adopted one method over the others and tend to force all software development activities to follow the adopted mode. Some of the more popular methodologies in use today include:

- Classic Life Cycle (Waterfall)
- Evolutionary (Prototyping/Iterative)
- Program Transformation
- Spiral Model

Mark Kross [29], Darrell Quick [44], and Peter Gordon [15] provided a thorough review of the Classic Life Cycle, Prototyping, and Iterative design paradigms. The general consensus was that the Classic Life Cycle does not lend itself well to the design of user interface systems. This is at least partially due to the limited dialogue between developer and user once the design starts. On the other hand, they found the prototyping and iterative methodologies to be well suited to user interface systems. Prototyping provides an avenue to communicate with the user to better determine the system requirements and to help prevent designing the wrong system. The Iterative Design methodology has the primary advantage that a working system is produced at each iteration. Thus, user interface capabilities and improvements can be incrementally added to the system.

The Program Transformation methodology is emerging as an attractive alternative to program generation. In this methodology, a formal specification of the system requirements is produced. This formal specification is then automatically transformed into syntactically correct code. Some human intervention may be required to assist in the transformation. The generated code is then validated aga nst the user's requirements. If the system must be modified, then the adjustments are made to the formal specifications and the process is repeated. With this method, there is no design stage whatsoever. Since a large part of user interface generation is developing the screen layout, this system is not the most desirable. This is because many screen designs can be produced from the same set of requirements.

The Spiral Model was developed at TRW as a risk-driven approach to the software development process [6]. In this methodology, the following steps are repeated until the program is fully developed:

- Determine objectives, alternatives, and constraints.
- Evaluate alternatives.
- Identify and resolve risks.
- Develop and verify the next-level product.
- Plan next phases.

Depending on the identified risks, the fourth step listed above may use the classic life cycle, prototype, iterative, or transform approach. Barry Boehm writes [6:65]:

> The spiral model also accomodates any appropriate mixture of a specification-oriented, prototype oriented, simulation-oriented, automatic transformation-oriented, or other approach to software development, where the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks, and the relative effectiveness of the various techniques in resolving the risks.

3.2 Combined Methodology

It is sometimes the case that no one methodology is best for a software design project. Gordon and Quick each decided on a hybrid paradigm for their designs. They combined

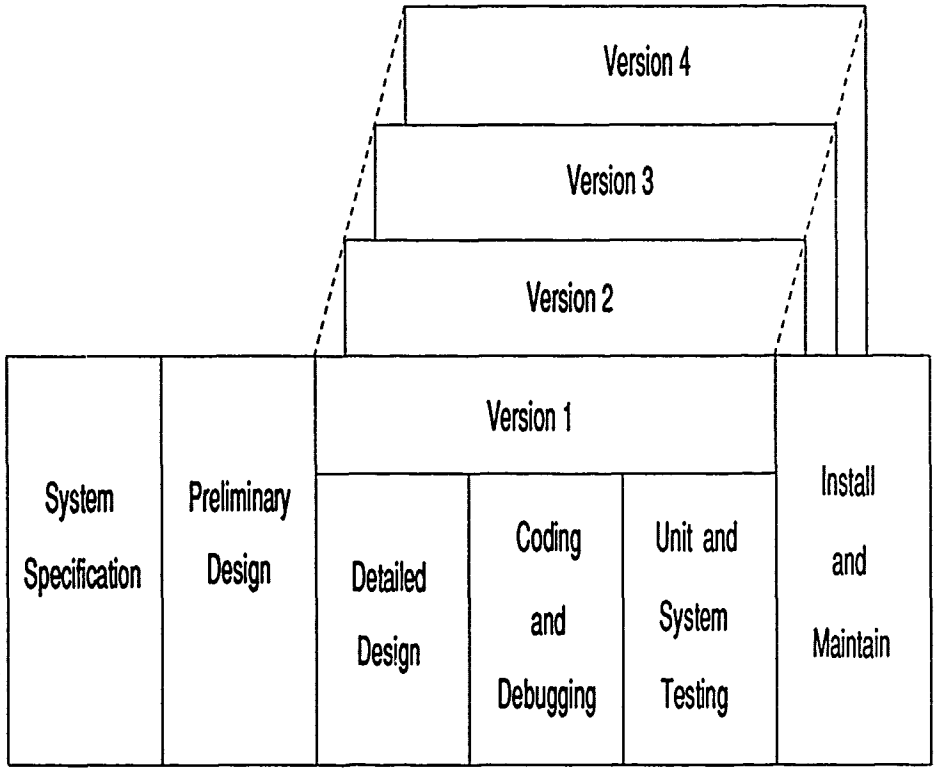


Figure 8. Darryl Quick's Design Methodology

the best characteristics of the classic life cycle, prototype, and iterative methodologies to produce their paradigms.

Quick's model, as depicted in Figure 8, begins with requirements gathering using prototypes to establish the requirements specification. A high-level preliminary design was then produced. This was followed by a detailed design, coding, and testing in an iterative manner. After the last version was produced, the system entered the installation and maintenance phase. Gordon's method was similar, but used prototypes as a part of the iterative detailed design, coding, and testing phases.

The combined methodologies developed by Gordon and Quick appeared to have considerable merit for the design of user interfaces. As such, the methodology used for the

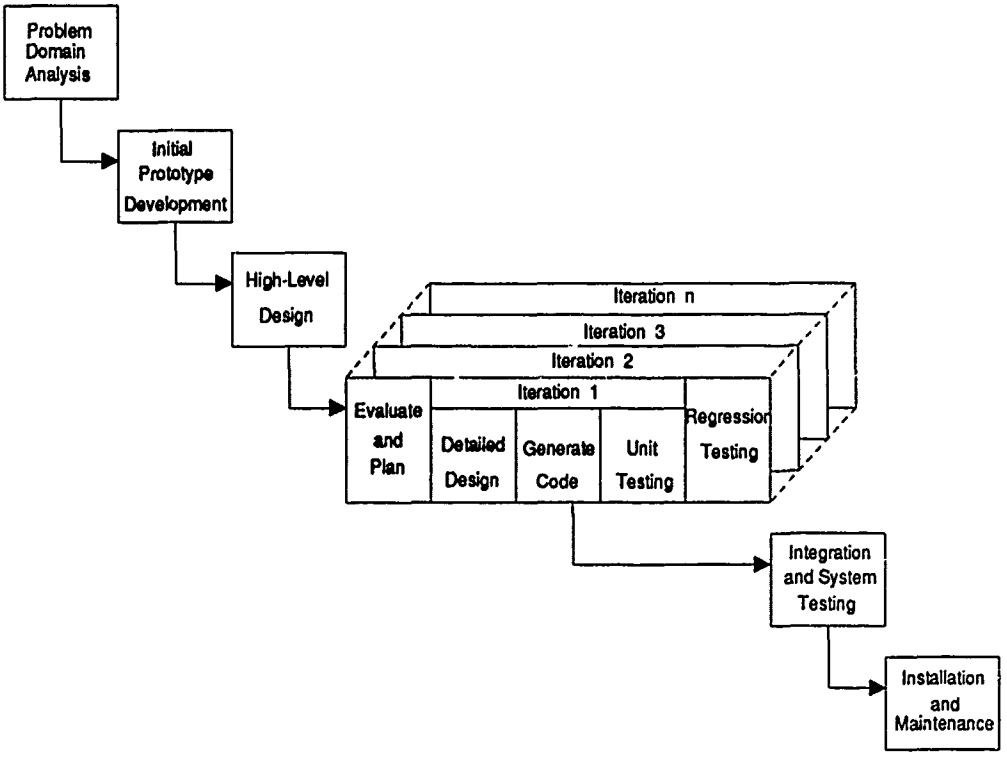


Figure 9. Saber User Interface Design Methodology

Saber user interface, illustrated in Figure 9, followed the general outline of their method with a few notable additions.

The first addition involved an explicit domain analysis phase before developing the initial prototype. Before jumping into a software design, the software engineer should have an understanding of at least the fundamental terms and concepts of the problem domain. If the software engineer has *current* experience in the domain area, this step may be omitted. However, if the designer has little or no experience or if recent advances have occurred in the field, a study or review of current topics should be conducted. Accordingly, the domain analysis conducted for this user interface consisted of:

- Reviewing recent thesis efforts in the area of wargames.
- Reviewing recent thesis efforts in the area of designing user interfaces for wargames.

- Conducting a literature review in the areas of user interfaces and the X Window System.
- Participating in numerous meetings with others involved in wargaming research.

After conducting the domain analysis, initial prototypes were developed. These consisted of sample screen layouts produced in a storyboard manner to show the expected behavior of the system. Accompanying each sample screen was a description showing the function, control flow, input fields, integrity constraints, and processing as was suggested by Kross [29]. In addition to the sample screens, several sample reports were produced to be reviewed and critiqued by the Air Force Wargaming Center personnel. The prototype stage was followed by an object-oriented high-level design of the overall user interface.

The remaining additions in the new methodology were part of the iterative stage. Before conducting the detailed design for each iteration, an evaluation and planning stage was conducted. This consisted of evaluating various alternatives for implementing the next portion of the high-level design. For each alternative, the expected benefits and risks were weighed. In addition, any constraints placed on the user interface were considered for possible impact. A plan was then developed for implementing the selected alternative in a manner that would maintain or enhance the object-orientedness of the overall design. Efforts were made to reuse as many components as possible while maintaining high cohesion and low coupling between modules and iterations. Using the appropriate plan, a detailed design was created. This, in turn, was used for generating the code for the given iteration. Unit testing was then performed on the individual module to ensure the proper functionality was achieved. Any time a major change or modification is made to a piece of software, there is the possibility of introducing errors into previously correct code. Thus unit testing was followed by regression testing to ensure the newly added module did not adversely affect the functionality of previously written modules.

After the last iteration was complete, the post-processor user interface software was integrated with the air and land simulations, the database software and the input interface to form the combined Saber theater warfare simulation. Installation and maintenance of the combined system was left to the Air Force Wargaming Center.

3.3 Summary

This chapter presented a summary of design methodologies currently being used for software development. This was followed by a description of the hybrid methodology used by Gordon and Quick. Lastly, the methodology chosen for Saber's post-processor user interface was described. This basically iterative methodology was a modified version of Gordon and Quick's method with an explicit domain analysis step added. Each iteration consisted of a risk analysis, detailed design, coding, unit testing and regression testing. The next chapter covers the requirements for the display screen and reports as well as the high level and detailed designs of the user interface.

IV. System Requirements and Design

Wargames and their user interfaces have been around for many years. Every wargame, be it a board game or a computerized simulation, has some form of user interface. This is because the game's players must have some idea of the location and status of forces in order to effectively play the game. In fact, the user interface plays an important role in the success and acceptance of the wargame. People will refrain from playing a game if they do not understand what is happening or if they find it difficult to control their forces.

Because of this importance, most user interface designers seek to display information to the game player in a way that is easy for the user to understand and control. However, deciding what to display and how best to display it is not an easy task. It is often the case that the user or designer may not have a solid understanding of what the game should look like before the user interface is written. A common method for overcoming this lack of understanding is through the use of prototypes.

This chapter first describes how prototypes were used to determine the system requirements. It then presents the high level and detailed design of the user interface. Since this project used an object-oriented design, the detailed design of each object or topic is presented individually.

4.1 System Requirements

The requirements and desired features for the Saber wargame were collected from a variety of sources. Some were obtained from wargames currently in use at the Air Force Wargaming Center or elsewhere. Others came from reviewing requirements for various wargames currently under development [14]. Still others came from discussions of the Saber research group and through the review of system prototypes.

4.1.1 System Prototypes. Two types of prototypes were used to determine the system requirements. The first consisted of using sample screen layouts which graphically depicted the possible appearance and behavior of the system. The second form of prototype

consisted of sample reports depicting the aircraft mission, airbase and land unit summaries to be provided.

4.1.1.1 Sample Screens. Most wargames with a graphical user interface have some common features. For example, it is common to display geographical regions and to use some character or symbol to designate the combatants. However, the specific representation of these and other items usually varies depending on the purpose of the simulation and on the intended audience. To gain a better understanding of the requirements for the Saber wargame, several sample screen layouts were generated using a commercially available, personal computer based graphics program. The sample screens were used to identify the following requirements:

- The user should have some method of determining the status of a base, unit or aircraft misssion.
- The user should have the capability to zoom in and out, as well as to pan across the display.
- The user should be able to specify which terrain features are displayed on the map.
- The user should be able to specify which units and bases are displayed on the map. For example, the user may want to limit the display to Blue airbases and aircraft missions.
- The user should have the capability to display the weather conditions on the map.

The sample screens also brought to light the issue of hexagon orientation. On the original sample screens, the hexagons which make up the map were oriented with the points to the north and south and the flats on the east and west sides. Figure 10 illustrates this hex layout. This is the orientation that was described in Mann's and Ness' thesis efforts. However, after some discussion, it was decided that the points of the hexagons for the Saber wargame would be on the east and west sides with the flats on the north and south as shown in Figure 11. The primary reason for choosing this orientation was that most of the wargames currently in use or under development at the Air Force Wargaming Center orient the hexagons with the points on the east and west.

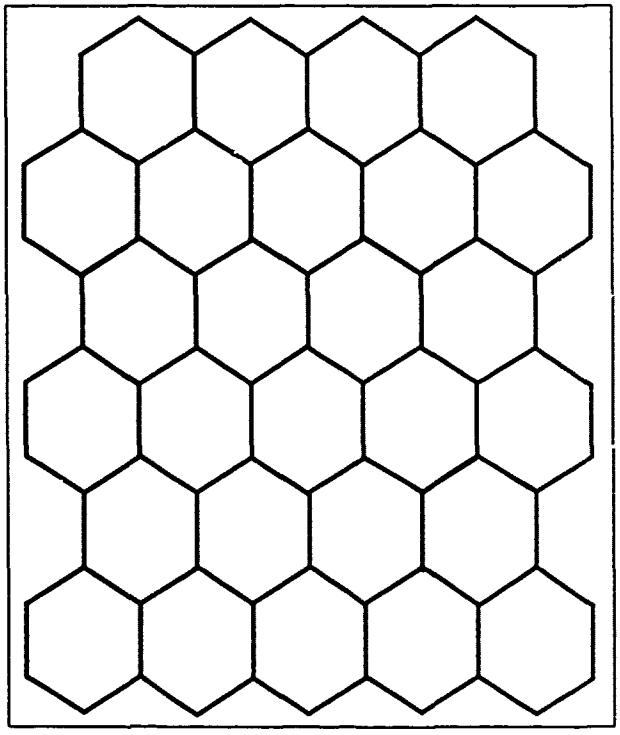


Figure 10. Mann's and Ness' Hexagon Orientation

A secondary reason for orienting the hexagons with the points on the east and west sides concerns the layout of information within the hex. Having the points on the east and west results in the hex being slightly wider across the center. Thus, a longer text string, such as a city name, can be placed in the center of the hex. Also, with this orientation, the flats are on the north and south sides of the hex. This provides two convenient locations to place the hex number; either just below the top flat or just above the bottom. With the hexes oriented the other way, the hex numbers must be shifted towards the center. This takes away some space for drawing other objects in the hex. Figure 12 illustrates these points.

Captain Mann described the hexboard as consisting of seven layers or planes of hexagons. Layer one corresponds to the ground level and the other six layers correspond

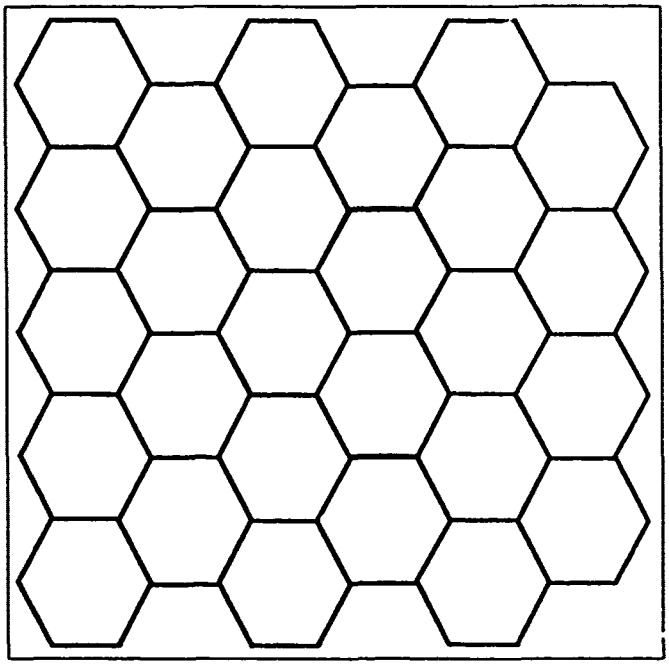


Figure 11. Saber Hexagon Orientation

to different altitudes above the ground layer. Layer one is composed of individual hexagons which measure 25 kilometers from side to side. Layers two through seven are composed of air hexes. An air hex is made up of the equivalent of seven ground hexes arranged as six hexes surrounding one center hex. Figure 13 illustrates this concept.

Each hexagon is identified with three numbers representing its location in the X, Y, Z coordinate space. The first number represents the layer in which the hexagon is located. The second two numbers correspond to the X and Y coordinates of the hex. Numbering of hexagons begins in the lower left corner of the hexboard. The X coordinate of the hexes increases as you go from left to right. Similarly, the Y coordinate increases from the bottom of the hexboard to the top. Figure 14 also shows the numbering scheme for the ground hexes. In this figure, the Z coordinate is not explicitly listed.

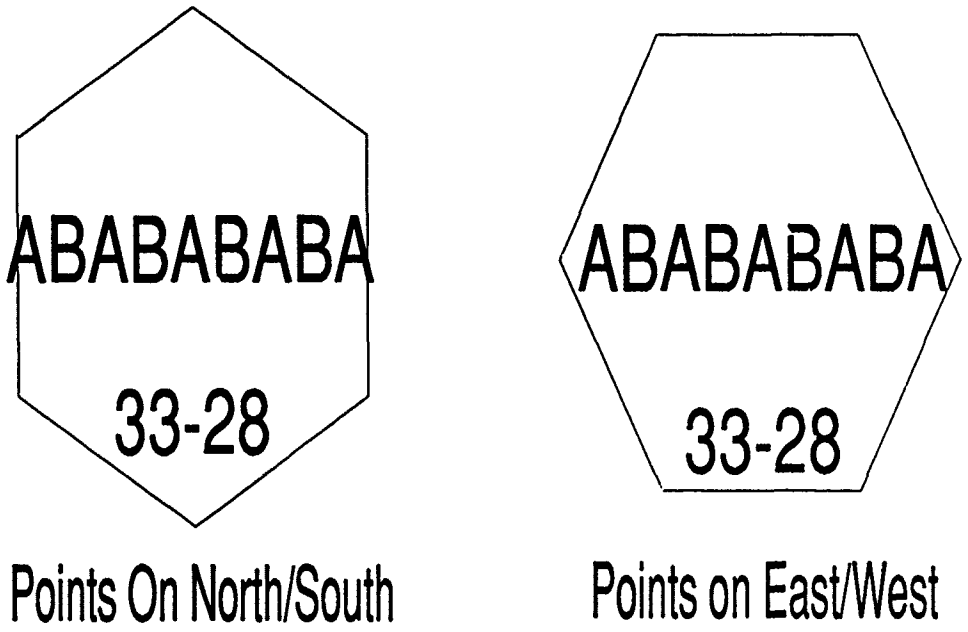


Figure 12. Comparison of Hexagon Layouts

4.1.1.2 Sample Reports. Determining what information to provide to the game players is not always easy to do. While it can be difficult, if not impossible, to anticipate all the needs of a particular individual or team of players, the system designer must make enough information available in order to satisfy the needs of as many potential users as possible. However, these needs may vary depending on what the player's expect to learn from playing the game. Thus, the system designer must also be careful not to restrict the scope of the reports that are provided.

With this in mind, the sample reports for the Saber wargame were designed with an emphasis on providing an abundance of information to the game participants. A requirement was then placed on the user interface to allow the players to select a standard subset of the reports to be generated each day. Furthermore, the user interface was required to

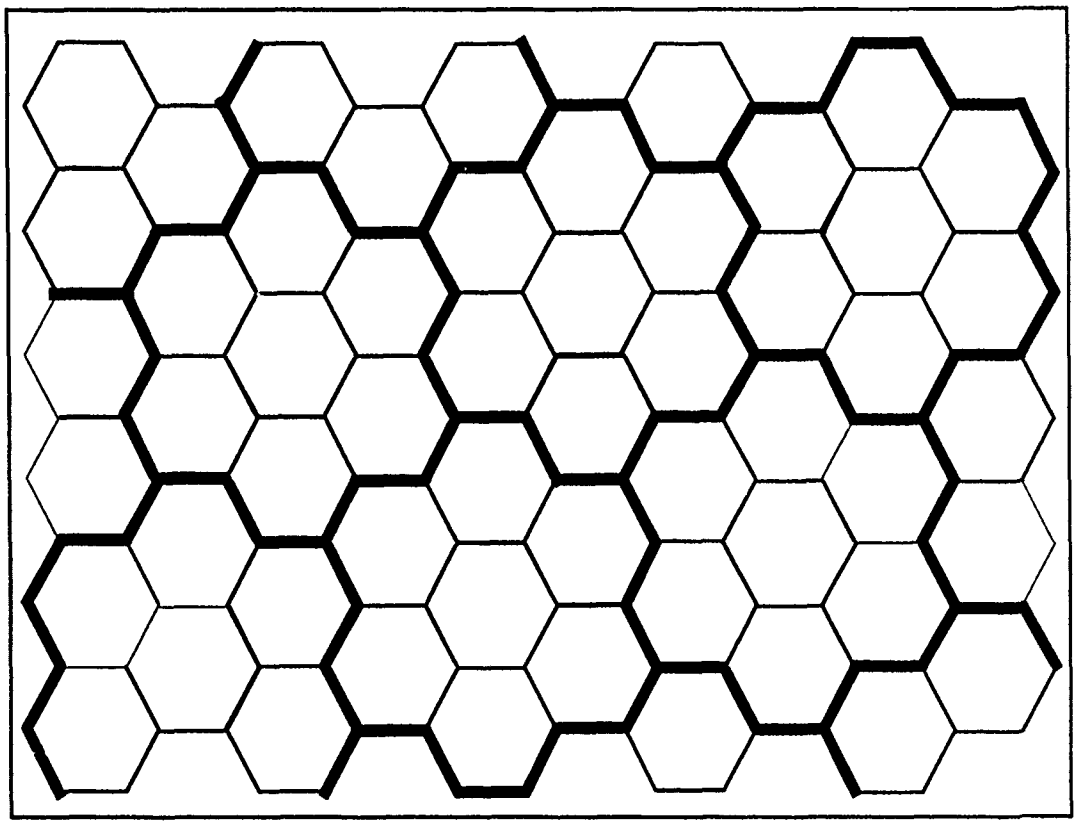


Figure 13. Saber Air Hexes

allow any report to be viewed or printed from the terminal at any time without the player having to memorize any operating system commands.

In an effort to increase the portability of the Saber wargame, a requirement was imposed that all reports would be restricted to a maximum width of 80 columns. Although many sites have printers capable of printing up to 132 characters on a line, designing reports for these wide carraige printers could cause problems if the game is run at a location with only the narrower printers.

Another requirement of the reports was to provide both daily and cumulative reports. The daily reports are useful for showing the status of forces and resources at a specific point in time. The cumulative reports, on the other hand, are useful for determining trends. For example, cumulative reports can show such things as the percentages of aircraft

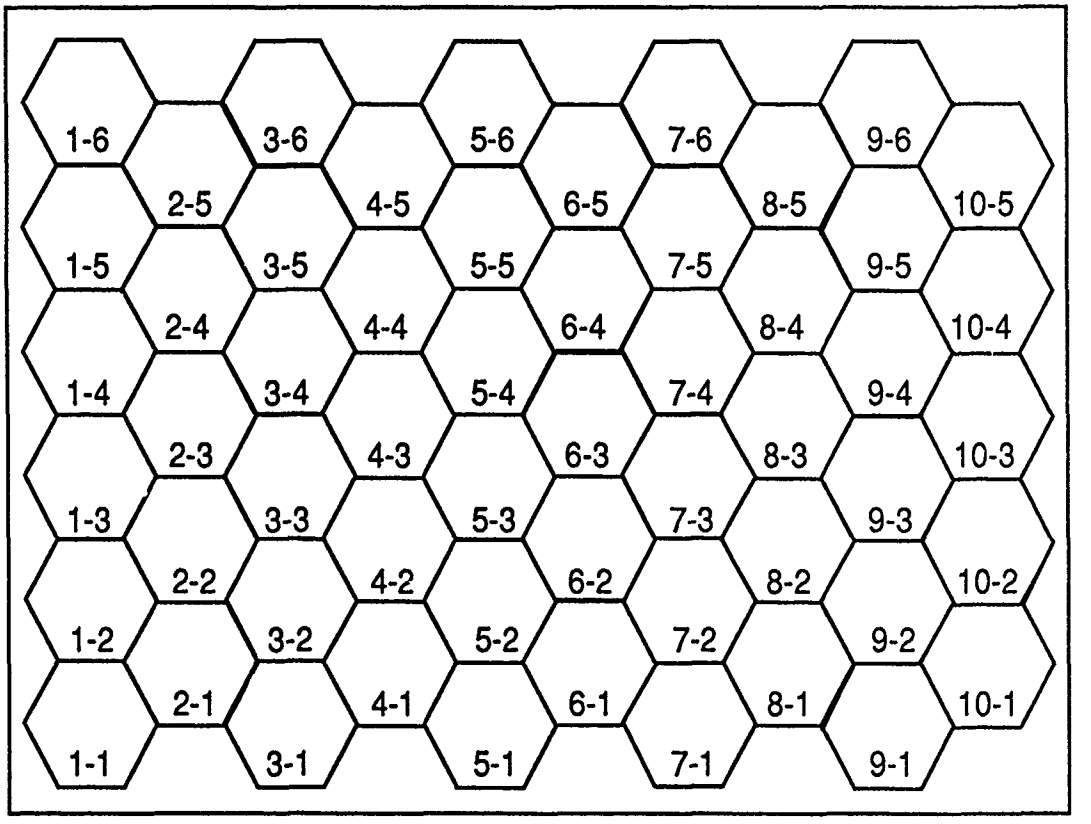


Figure 14. Saber Hex Numbering Scheme

apportioned to various types of missions, or the total aircraft lost by mission type during a campaign.

The sample reports also raised the issue of whether the mission input reports should be sorted by mission number or by target number. The latter was chosen because it allows the players to scan the reports to see what assets have been assigned to each target.

4.1.2 Data Retrieval. Data to be used by a simulation or wargame is typically stored in a database, in files, or in a combination of the two. Retrieval of data from a database can usually be accomplished fairly easily using the methods provided by the database software. For some of the more advanced database packages, this can be accomplished using such methods as a Structured Query Language (SQL) or query by example. Depending on the language used to write the wargame, the queries can sometimes be

embedded in the applications software. This can greatly simiplify data retrieval for the simulation.

However, if the data used by the wargame is stored in or retrieved from a database, then the database software must physically reside on the machine which is executing the wargame. Since there are several quality database packages in use today, writing a simulation that relies on a particular database system could limit the number of machines and locations that could be used to execute the wargame.

Because of this, all data for the Saber wargame is stored in "flat" files. These files are constructed to mirror the relations that a database would use to store the data. While the use of these files makes the wargame more portable, it also increases the complexity of data retrieval. Routines must be written to fill internal data structures with information read from the files. Because the data is normalized, the information for a particular entity may be spread out over several files. Every record in a file is uniquely identified by one or more fields designated the key fields. It is these key fields that tie the information together from two or more files. Thus, when loading the internal data structures, the routines must be written to search through the files to find the entry with the matching key field. Once the record is found, the software must extract the desired information from the appropriate fields.

4.2 Saber Design

The design of a software system is one of the most important parts of a software development effort. Extra effort spent designing a system can reduce the number of errors generated in the coding phase of a project. Of the many design styles available, object-oriented design lends itself particularly well to user interfaces using the X Window System and the Motif widget set. The Motif widget set, itself, is composed of object classes and a hierarchy of objects (widgets) that inherit attributes from objects above them.

A high level object-oriented design was accomplished for the Saber user interface after the system requirements were determined. This entailed identifying the objects, their attributes and the methods (or actions) which operate on the objects. After the

high level design was completed, the interaction between the user interface and the X Window System was formalized. This, in turn, was followed by a detailed design and implementation of each object in an iterative manner as was described in the last chapter.

4.2.1 High Level Design. The first step in producing the object-oriented design was to identify the major objects and object classes. Booch defines a class as a set of objects that share a common structure and a common behavior[7:93]. An object is then an instance of a class. In a user interface, everything that is displayed can be considered as an object. Thus, a good idea of what the actual display would look like was needed. The sample screens used for requirements gathering proved extremely useful in this regard. The main display consisted on a window with a menu bar across the top and a large map underneath. The map was composed of hexagons which contained various terrain attributes as well as land units and airbases. Additionally, Ross lists the following as sources of potential objects[45:9]:

- People: humans who carry out some function
- Places: areas set aside for people or things
- Things: physical objects, or groups of objects, that are tangible
- Organizations: formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals
- Concepts: principles or ideas not tangible *per se*; used to organize or keep track of business activities and/or communications
- Events: things that happen, usually to something else at a given date and time, or as steps in an ordered sequence

The object classes identified from the screen prototypes are shown in Figure 15 in a manner similar to what Booch suggests[7]. They were classified as either application or Motif object classes. The application classes, described in Table 1, are those things that were unique or specific to the Saber wargame. The Motif object classes are container widgets composed of a group or groups of child widgets. The purpose of these object classes is to consolidate the routines necessary for the creation and modification of the particular objects. One way in which these objects are modified is through the addition of its children. The Motif object classes and their descriptions are shown in Table 2.

Table 1. Application Object Classes

Object	Description
Game Player	The game player is the person playing the wargame.
Reports	Reports are statistical summaries presented to the game players.
Airbases	Airbases are air force bases belonging to the Red, Blue, or neutral forces.
Land Units	Land units are army ground units belonging to the Red, Blue, or neutral forces.
Aircraft Missions	Aircraft missions are collections of aircraft grouped to accomplish a specific task. They may be composed of aircraft from Red, Blue, or neutral airbases.
Terrain	Terrain features are natural or man-made assets located in or on a hexagon. They include such things as rivers, roads and cities.
Hexboard	A hexboard is a collection of hexes arranged in a rectangular pattern.

Table 2. Motif Object Classes

Object	Description
Menubar	A menubar is a collection of pulldown menu widgets associated with topics on a rectangular widget across the top of the display area.
Toggle Board	A toggle board is a collection of toggle button widgets along with OK, CANCEL, and HELP pushbuttons.

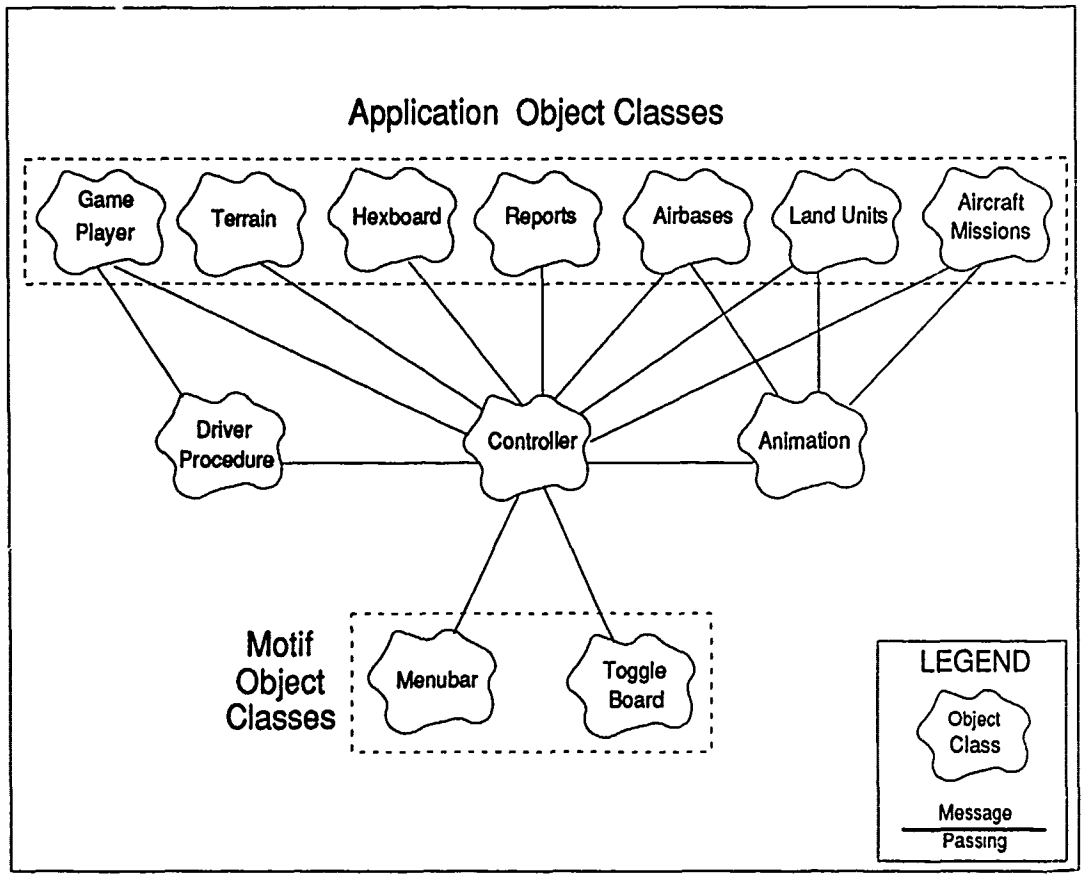


Figure 15. Saber User Interface Object Diagram

After the major classes were identified, the next step was to specify the class attributes. Many of these attributes were needed for the creation and manipulation of the various objects. Others are features needed for producing the graphical objects on the screen. The database relations produced by Capt Andre Horton[20] were the primary source for the attributes of the application objects. The attributes for the Motif objects consisted of the Motif resources applicable to the particular objects. These included such things as the width, height, margins and offsets of the child widgets.

Identification of the attributes was followed by a listing of the methods or actions to be provided for each object class. Initially, these lists did not include any parameters. The intent was simply to document what actions could be performed on the objects. Once this was done, the methods were closely analyzed and the required parameters were determined.

While identifying the methods and parameters, the reasoning and justifications for each decision or choice were documented. This proved to be quite useful when coming back to the object classes to complete the detailed design and implementation. Following is a more detailed description of each class:

- Application Object Classes
  - Game Player: This class has methods for setting and retrieving the player's side, the seminar number, and the current day.
  - Reports: This class has methods for setting a standard set of reports, printing reports, and viewing reports.
  - Airbases: This class has methods for creating, displaying, and erasing airbases; and for displaying and erasing status boards.
  - Land Units: This class has methods for creating, displaying, and erasing land units; and for displaying and erasing status boards.
  - Aircraft Missions: This class has methods for creating, displaying, and erasing aircraft missions; and for displaying and erasing status boards.
  - Terrain: This class has methods for reading, displaying, and erasing terrain information. The terrain information consists of trafficability, roads, rivers, railroads, obstacles, Forward Edge of Battle Areas (FEBA's), country borders, coast lines, pipelines and cities.
  - Hexboard: This class has methods for the creation and deletion of hexboards.
- Motif Object Classes
  - Menubar: This class has methods for the creation of menubars and for the addition of pulldown menus.
  - Toggle Board: This class has methods for the creation and deletion of bulletin boards that contain toggle buttons. It also has methods for the creation and deletion of data structures to be passed as parameters for the board creation.

The object classes just described were the basic building blocks used to create the user interface. However, the classes by themselves are not useful until objects are instantiated. Objects can be instantiated by another object or by some controlling module. In many programs, this controlling module is referred to as the main driver procedure. Unfortunately, this design does not work very well for programs which use the Xt Intrinsics. The reason for this is that the main procedure, after performing various initializations, typically enters a main loop through a call to the *XtMainLoop* function. This routine is an infinite loop that retrieves and dispatches events from the X event queue. When an event is dispatched for which a callback has been registered, processing in the main procedure is suspended and some other subroutine is executed. These callback procedures are the ideal place for object instantiation to take place.

The application programmer has a range of options available when developing the callback procedures. At one end of the spectrum, an individual callback can be written for every event that the program needs to be made aware of. For example, each button on a pulldown menu can have its own callback procedure. At the other extreme, a single callback can be written that handles all events. If this method is used, the callback procedure must be able to determine what type of event triggered the callback. This can be accomplished by examining the event record created by the X Window System and passed as input to the callback procedure. Another way to determine what the event was is through the use of "client data" passed to the callback. This data is specified by the programmer when registering callbacks with the system. The client data can be of any type, and used for any purpose, that the programmer wishes. Thus, the client data could be used to identify why the callback procedure was entered.

When designing the arrangement of the callbacks, it is important to take into consideration the Motif widget hierarchy. The widgets used in an application program can be arranged in a hierarchy with all widgets, except for the top level widget, having exactly one parent widget. The widget id of the parent must be specified whenever a new widget is to be created. Thus, if a procedure is to create a new widget, it must have access to the widget's parent. One method of obtaining the parent's widget id is to receive it as an input parameter. By default, callback procedures always receive a parameter specifying

the widget for which the callback was registered. If this widget is not the desired parent, then another alternative is to pass the parent's widget id as client data. Unfortunately, the parent's widget id may not be known at the time when the callback was registered. In this case, the only alternative is to make the parent widget globally available to the callback procedure. This, then, suggests that the callback procedures be grouped such that all needed parent widgets are visible. It makes sense to keep the group of global variables and associated callbacks as small as possible.

For the Saber user interface, several of the instantiated objects are either widgets themselves or require access to certain widgets. Since the selection of items on the menus presented to the user often involve the creation or manipulation of these objects, the design called for one or more controller packages that contained the callback procedures for each menu item. If the menu items can be separated into groups such that each group of associated callbacks deals with a single or small set of widgets and objects, then it is sometimes possible to develop a separate controller package for each group.

For the Saber wargame, it was not possible to group the menu items in the manner just described. Therefore, a main driver procedure and a controller package were added to the application and Motif objects in Figure 15. It is important to realize that the Saber design is still object-oriented. The issue is when and where the objects are to be instantiated. A complete description of the object classes, their attributes and their methods is given in Appendix A. The high level design of the was followed by a formalization of the relationship between the user interface application software and the X Window System libraries.

4.2.2 Interface to the X Window System. Some user interfaces can be implemented by simply calling subroutines in the Xt Intrinsics and Motif widget set. Others may require additional calls to selected Xlib routines. The Saber user interface fit into the latter category. Displaying the graphical symbols for the airbases, aircraft missions, and land units required the use of low-level Xlib subroutines to create the appropriate pixmaps. Adrain Nye defines a pixmap as an array of pixel values that is stored in memory until it is copied into an existing window [38].

Due to the need to access the Xlib, Xt Intrinsics and Motif libraries, it was clear that, as a minimum, the SAIC bindings would have to be used. The choice remained of whether to supplement it with the Boeing bindings or the Ada/Xt software developed by Unisys. Using the Ada/Xt software would have required the full or partial development of an Ada implementation of the Motif widget set. The Boeing software, on the other hand, already had bindings developed for Motif. Thus, the decision was made to utilize the Boeing bindings in combination with the SAIC software to develop the Saber user interface.

The Saber user interface was also designed to use a hex widget designed by the Air Force Wargaming Center. This object-oriented widget contains routines to create and manipulate hexboards. Routines are provided to display certain features inside of a hex. These features include rivers, roads, cities, city names, forestation, and background color. Since the Saber user interface displays a few more hex attributes, some modifications to the hex widget were required.

According to the documentation, "The hex widget is a constraint widget which allows child widgets to be mapped inside hexes"[17]. This means that the symbols for airbases, aircraft missions and land units could be created as pushbutton widgets and assigned to particular hexes. The hex widget keeps track of how many widgets (unit/base/mission symbols) are assigned to each hex. If more than one is widget is located in a hex, the hex widget displays a "stacking dot" to alert the game players.

Since the hex widget is written in the C programming language, Ada bindings had to be developed. These hex bindings were modeled after Boeing's bindings to the Motif widget set. The mapping from Ada to C was fairly straightforward. No internal Ada data structures were needed. In fact, most of the bindings used only the address of the hexboard and numeric values for such things as the width, hex sides, and colors.

The relationship between the Saber user interface and the various Ada bindings is shown in Figure 16. The figure accurately reflects that the Boeing software contains bindings to a small subset of the Xlib functions in addition to the bindings to the Xt Intrinsics and Motif widget set. The user interface may make calls to the Boeing bindings, the

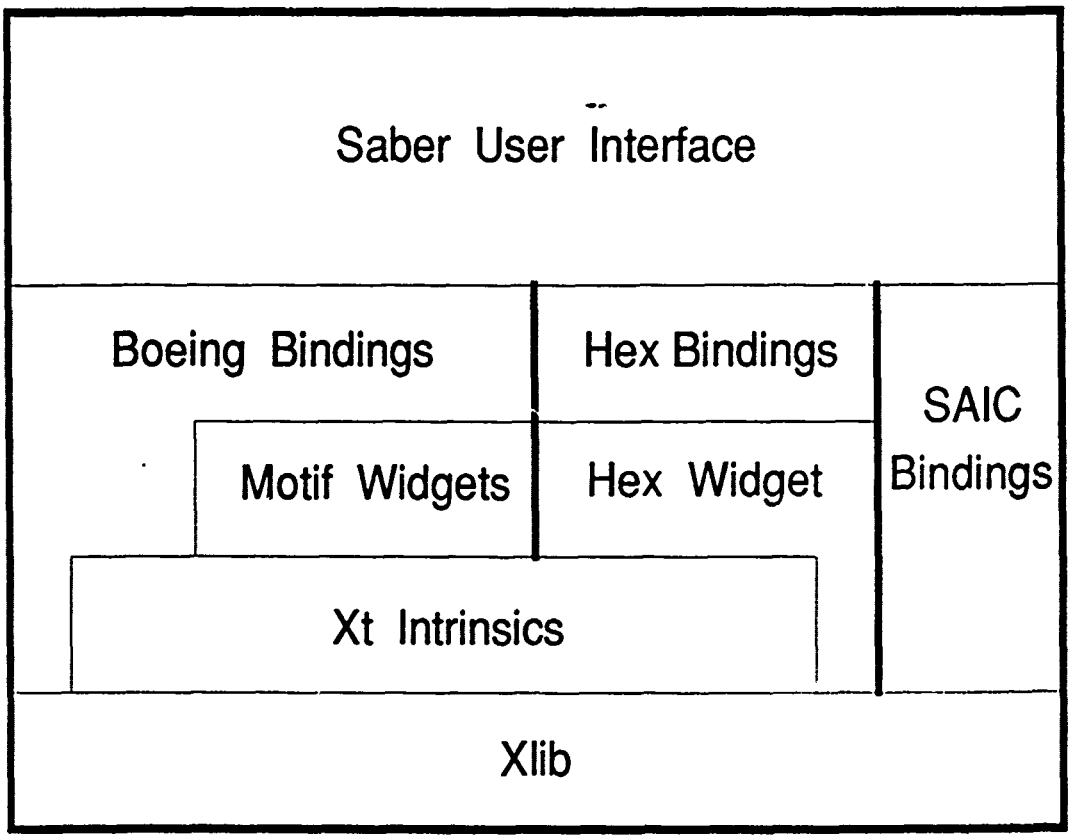


Figure 16. User Interface Relationship to the Ada Bindings

SAIC bindings, and the hex widget bindings. In fact, interactions between the application program and the X Window System are made solely through these bindings.

4.2.3 Detailed Design. The detailed design phase involved an in depth analysis of each object and interface feature to determine the best design. The goal was to design the objects in sufficient detail to allow for effective implementation. An overriding concern was to maintain or increase the object-orientedness of the high level design. Since it was known that the user interface would be implemented using the X Windows System, it made sense to design the objects to take advantage of the capabilities provided by Xlib, the Xt Intrinsics, and the widget sets.

The following sections describe the detailed design issues for the Saber user interface.

4.2.3.1 Main Menu Bar. An important part of a user interface is the design and placement of the primary system controls. The layout of the controls depends largely on the nature of the application. Some of the more popular approaches are to use a control panel, a menu bar, or pop-up menus [41].

A control panel is appropriate for grouping like controls that have similar functions. Control panels are commonly used in commercially available graphics packages such as DrawPerfect or Dr Halo.

A menu bar is a horizontal bar located just below a window's title area. It contains a list of menu topics that can be selected with a mouse or through the keyboard. When an item on menu bar is selected, a pull-down menu appears with selectable items related to the chosen topic.

A pop-up menu has many of the same features as a pull-down menu. Pop-up menus may appear whenever the sprite is in a window and the user clicks on the left mouse button. Their primary advantage is that no mouse movement is required to generate the menu. Another advantage is that no space on the screen is required until the menu is activated. However, this can also be a disadvantage in that the user may have no idea that the pop-up menu even exists.

Which of the methods to use is a decision the application programmer must make. However, the *OSF/Motif Style Guide* notes that, "Because menus are a principal method of interaction between users and OSF/Motif applications, most applications require a menu bar"[41:4-3]. The advantage to using menu bars is that the user has a visual cue of what topics are available. The user can easily browse through the pull-down menus on the menu bar to see an informal "table of contents" for the interface.

In keeping with OSF/Motif guidelines, the Saber user interface was designed using menu bars. The main menu bar is shown in Figure 17. The pulldown menu hierarchy is displayed in Figure 18.

4.2.3.2 Hexes and Terrain. The hex widget written by the Air Force Wargaming Center was an integral part of the Saber user interface. Originally, the hex widget provided procedures for adding roads, rivers, cities, and city names to individual hexes.

Figure 17. Saber Main Menu Bar

Modifications to the hex widget were necessary to include procedures for adding railroads, pipelines, country borders, FEBA lines, air hex borders, bridges, and minefields to the hexes.

The assets for each hex can be classified as center, radial, and hex side assets. Center assets are assets drawn in the center of a hex. They consist of cities and city names. Figure 19 illustrates a hex with center assets. Radial assets are assets drawn from the center of a hex to a hex side. The radial assets are roads, railroads, and pipelines. Figure 20 shows a group of hexagons with various radial assets. Hex side assets are drawn along the side of a hex. They consist of rivers, country borders, FEBA lines, air hex borders, bridges, and minefields.

The hex widget maintains the asset information about each individual hex in its internal data structures. When the hexes are drawn on the screen, the hex widget treats each hex as an individuial entity and only draws assets inside of the hex. For center assets and radial assets this creates no problem. However, this does create an interesting situation for the hex side assets.

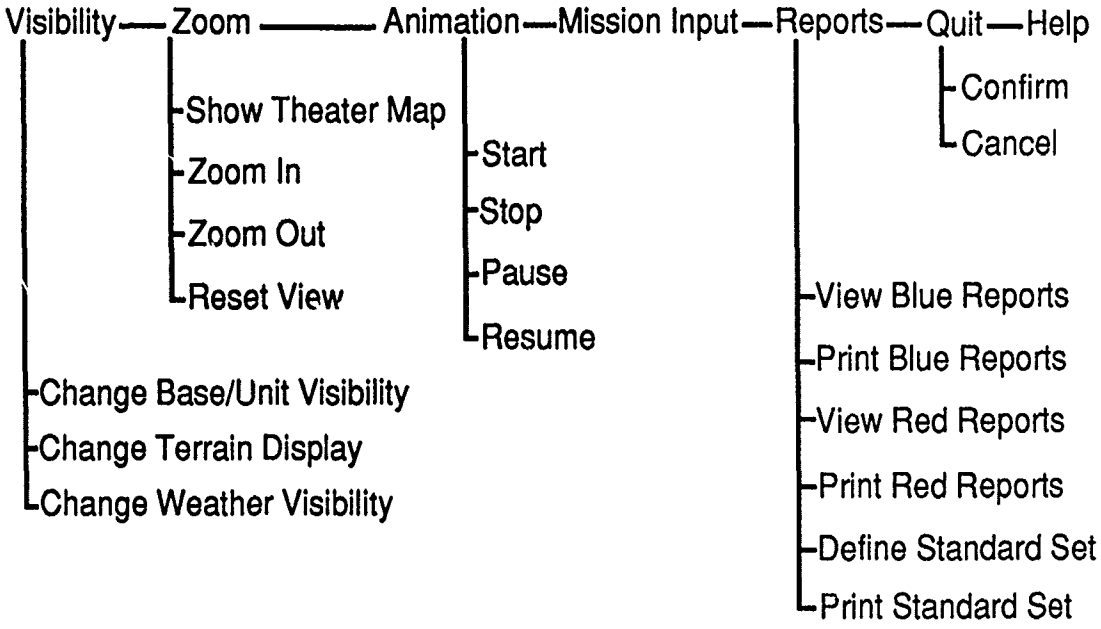


Figure 18. Saber Menu Hierarchy

The hex side assets are designed to be drawn along a line segment which is shared by two hexes (except for the hex sides that form the outside edge of the hexboard). Since the hex widget only draws inside of a hex, this means each hex draws one half of the side asset. Thus, as illustrated in Figure 21, to draw a river between two hexes, a river segment must be added to each of the hexes. The hexside to receive the river corresponds to the side that the hexes have in common.

For any given hex, there is always the possibility that there will be more than one type of radial asset to be drawn to the same side or that there will be more than one type of hex side asset to be drawn along the same hex side. Thus, special consideration was given to the widths of the radial and hex side assets. The widths had to be assigned so that all hex assets are visible on the screen. The assigned widths (in number of pixels) for

Figure 19. Saber Center Assets

the radial assets are shown in Table 3 and the assigned widths for the hex side assets are given in Table 4.

Due to the sheer number of hex assets, the display screen will at times appear cluttered. Because of this, and the fact that the game players may not be interested in certain assets from time to time, the user interface was designed to allow the players to customize the terrain display. The user can select exactly which terrain features are displayed at any time.

The player indicates his choices by clicking on toggle buttons in a "Terrain Display Options" window that is activated from the main menu bar. Figure 22 shows the design of this window. Any time the user changes the toggle values and exits via the "OK" button, the hexboard is redrawn to reflect the updated values.

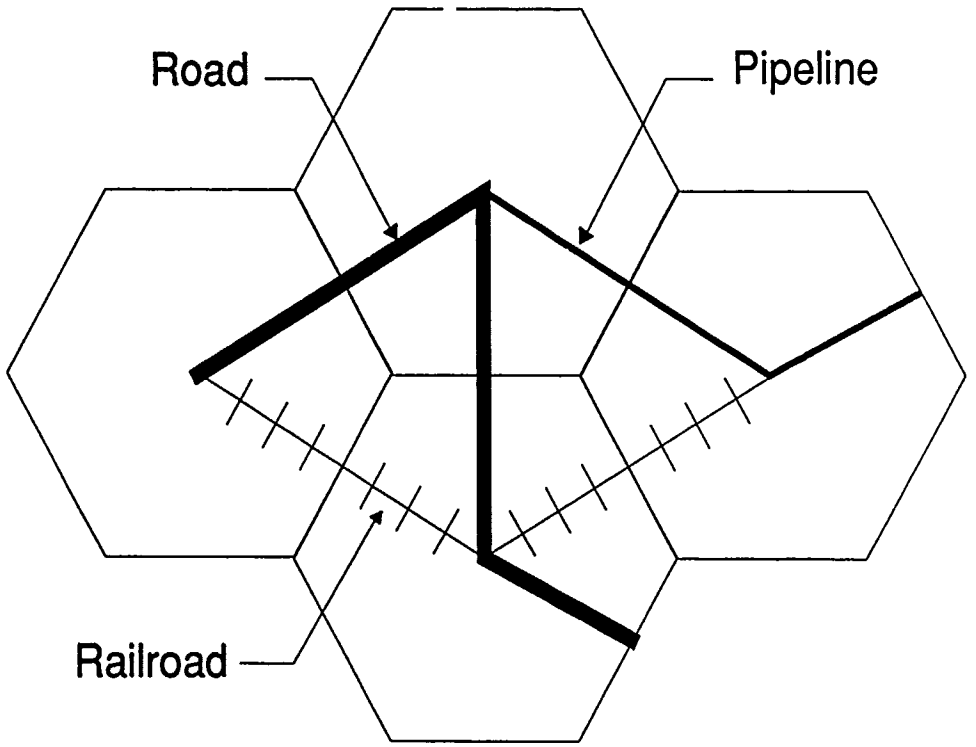


Figure 20. Saber Radial Assets

4.2.3.3 Help. The Motif widget set provides a variety of methods for presenting help to the user. One of the simplest involves putting a status/help bar at the bottom of the applications window. This method allows for a single line of text to be displayed at any time. Unfortunately, this method also takes away a portion of the screen that could be used for other purposes.

A second method is through the use of information dialogue boxes. These widgets can be made to appear when the user clicks on a "HELP" button or menu item. For example, Figure 23 illustrates a help screen for the "Terrain Display Options" board described in the previous section. The information dialog boxes can be customized to provide additional levels of help simply by adding a button to the bottom of the dialog box that displays additional or more detailed help.

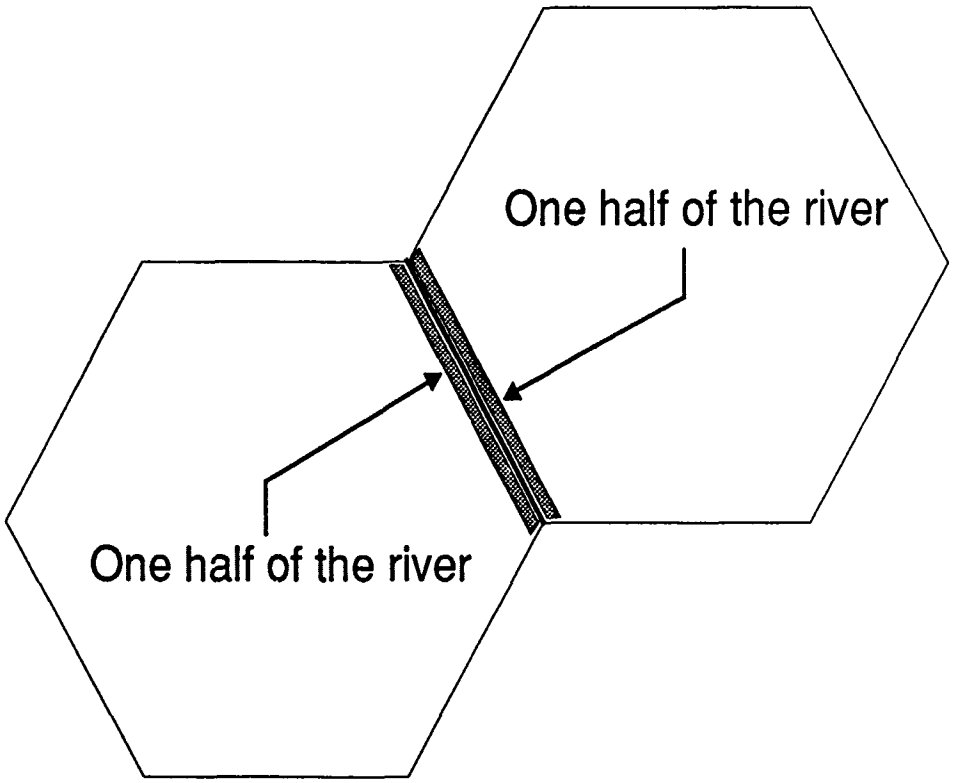


Figure 21. Saber River Segment

4.2.3.4 Weather. The hexes for the Saber wargame are organized into a number of weather zones. In general, all of the hexes in a single weather zone will be adjacent to each other forming a contiguous ares of the map. The weather zones and the hexes that they contain are read in from the database flat files. Each weather zone has a single type of weather which can be good (GD), fair (FAIR), or poor (POOR).

Representing the weather on the hexboard presented an interesting challenge. Darrell Quick used a weather overlay to represent weather zones for the Theater War Exercise[44]. He used different patterns and colors to represent each type of weather. While this method met the requirements for the simulation, it had the drawback of obscuring the underlying playing surface. The design for the Saber user interface had to overcome this problem.

The solution to the problem was found in the AFWC hex widget. The procedure

Table 3. Radial Asset Widths

Radial Assets	
Asset Type	Width (in pixels)
Road	7
Railroad	5
Pipeline	3

Table 4. Hex Side Asset Widths

Hex Side Assets	
Asset Type	Width (in pixels)
FEBA	7
Country Border	5
Air Hex Border	4
River	3

HX_SetHexBackground is used to specify the background color of each hex. When calling this procedure, the hex widget allows for the specification of a stipple. Douglas Young defines a stipple as "a repeating pattern produced by using a bitmap (a pixmap of depth one) as a mask in the drawing operation"[63:211]. The hex widget uses the stipple pattern to determine the color of each pixel for the hex background. Instead of drawing the hex background as a solid color, the hex widget only colors the pixels that correspond to bits set to one in the stipple pattern.

The background color used with the stipple pattern can either be the original background color of the hex or the original background color can be temporarily replaced with a new color that is unique to the type of weather. Using a new color, together with the stipple pattern gives the user two visual clues about the type of weather in a particular hex. However, if the stipple patters are distinctive enough, retaining the original background color is possible. The benefit to this approach is that the game player does not lose any information while the weather is being displayed.

While the Air Force Wargaming Center uses a stipple pattern to shade the Korean

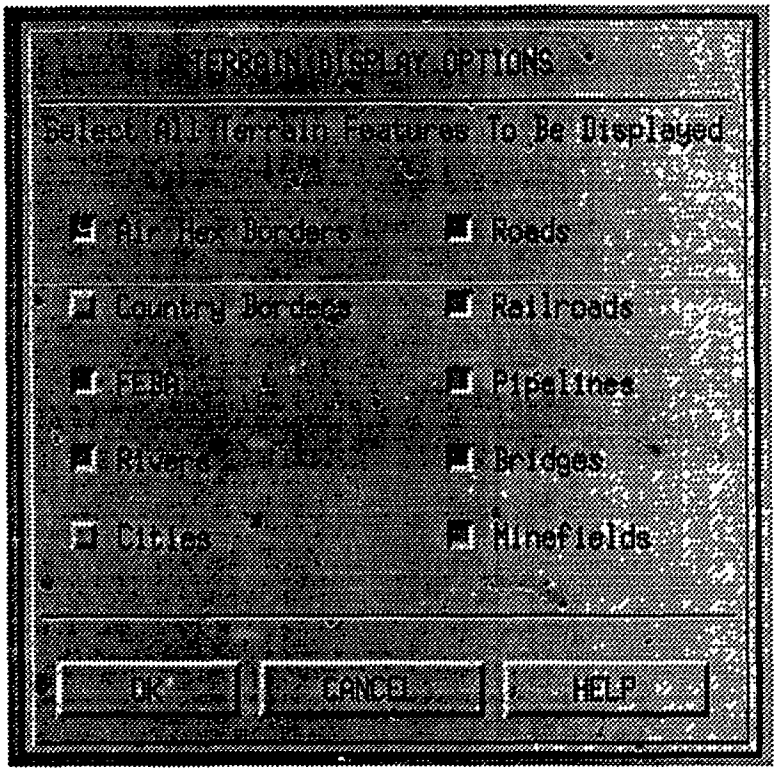


Figure 22. Terrain Display Options Bulletin Board

Demilitarized Zone in some wargame scenarios, this seemed like a logical way to represent the weather for the Saber user interface as well.

The decision of when to display the weather is up to the game players. The users may display and remove the weather display through the *Visibility* pull-down menu on the main menu bar.

4.2.3.5 Airbases, Land Units, and Aircraft Missions. The detailed designs of the airbase, land unit, and aircraft mission objects are all very similar. Each of these objects has the same general methods, but with somewhat different implementations. Treating the objects as separate classes makes it possible to satisfy the special requirements of each class. Also, the internal representation of the objects can be later modified, if necessary, with minimum impact to the other object classes. What sets the objects apart is the information stored, and the method and time of creation.

While airbases, land units, and aircraft missions each contain methods for updating

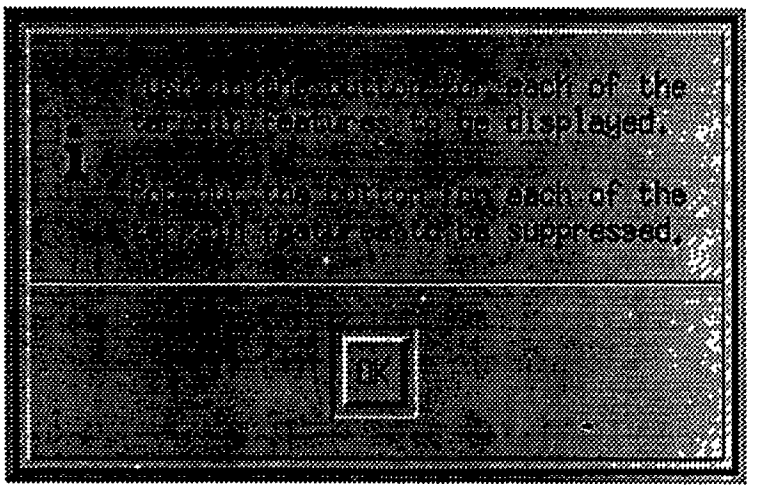


Figure 23. Sample Help Screen

and displaying their status, the actual information maintained and the format in which it is displayed varies significantly. Also, land units and aircraft missions can move during animation, whereas airbases cannot. Another distinction between the units concerns the longevity of the objects. Land units and airbases are relatively long lived in that they will have status information saved at the end of each game day unless they are completely destroyed. Thus, these objects can be created by reading the database flat files produced at the end of each day. Aircraft missions, on the other hand, typically do not exist for more than one or two days. The majority of the aircraft packages are created, fly their mission, and return to their bases with one game day. For this reason, most of the aircraft mission objects are not created until the animation is activated. The data used to create these missions is retrieved from the history file as the animation progresses as opposed to the flat files used for the land units and airbases.

One thing that is consistent among the objects is the representation of the objects on the display screen. Airbases and aircraft missions can each be represented by a particular graphical symbol. However, the specific symbols used to represent land units depends on the type and size of the unit. Army FM 101-5-1 provides a comprehensive list of the standard symbols used to represent ground forces [12]. Another thing that the objects have in common is that they are identified by a name. Because of size constraints, the length of the text string for the name should be held to under nine characters.

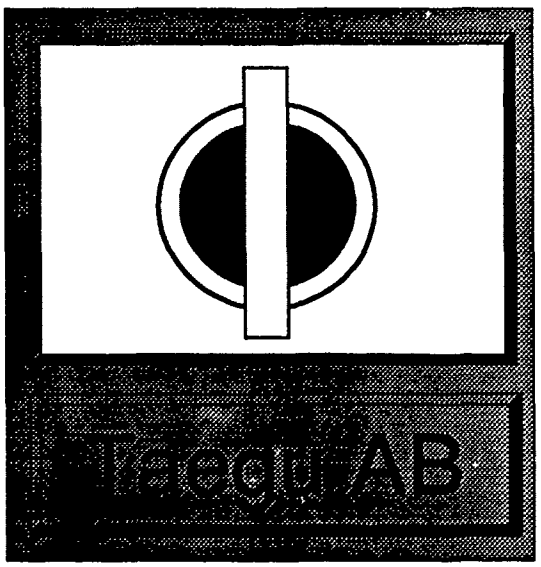


Figure 24. Sample Saber Airbase Representation

As was previously discussed, the hex widget allows child widgets to be assigned to each hex. Thus, the units are created as two pushbutton widgets arranged vertically in a Motif form widget[17]. The symbol for the unit is drawn on the top pushbutton using a pixmap read in from a file. The bottom pushbutton contains the name of the unit. Figure 24 provides an example of the widgets used to represent Taegu Airbase.

If more than one unit is located in a hex, the form widgets will be stacked on top of each other and only the top unit will be visible. Clicking on the unit symbol pushbutton will rotate the top unit to the bottom of the stack so that the next unit becomes visible. Clicking on the name pushbutton will open a window to display the status of the unit. The format and contents of the status window depends on whether the unit is an airbase, land unit, or aircraft mission.

4.2.3.6 Zoom and Pan Features. The user of a wargame often needs to view the hexboard at different levels of detail. Sometimes a high level view of the entire theater is needed. Other times, the game player may want to focus in on a particular area of the map. When the display is zoomed in, the player also needs to be able to pan horizontally and vertically in order to view different parts of the map.

The pan feature is easily accomplished by displaying the main hexboard inside of a Motif scrolled window widget. The scrolled window allows for horizontal and vertical scroll bars to be placed on two sides of the window. These scroll bars can then be used to pan across the display.

The zoom feature, on the other hand, can be accomplished in various ways. One method used by the Air Force Wargaming Center is to create two hexboards. One of the hexboards is larger than the other and is the primary window for displaying terrain features and unit symbols. The other hexboard is drawn using much smaller hexes. This theater overview map (or mini hexboard) shows the entire theater of conflict. A rectangular box is drawn on the theater map corresponding to the section of the map that is visible in the larger, primary window. The user can click the mouse while the sprite is in the theater map to move the rectangular box and correspondingly change the area of the map that is visible in the primary window. The Saber user interface was designed to use this feature to keep it consistent with the AFWC simulations.

The Saber user interface was also designed to provide a magnification capability. The user can increase or decrease the magnification of the hexboard by selecting the *Zoom* item on the main menu bar. The key to the magnification process is in the creation of the hexboard.

When the hexboard is created, one of the parameters to be specified is the hex radius. The hex radius is the distance, in number of pixels, from the center of a hex to a corner point. An increase (or decrease) in magnification can be accomplished by creating a new hexboard with a larger (or smaller) hex radius.

4.2.3.7 Color. The assignment of colors to the various drawn objects is a matter of personal taste. While the colors can be chosen by the application programmer,

Table 5. Main Hexboard Customizable Objects

Red Airbase Foreground	Red Ground Unit Foreground
Red Airbase Background	Red Ground Unit Background
Blue Airbase Foreground	Blue Ground Unit Foreground
Blue Airbase Background	Blue Ground Unit Background
Red Unit Name Background	Hex Outline
Ocean	Stacking Dot
River	Excellent Hex Trafficability
Stream	Very Good Hex Trafficability
Road	Good Hex Trafficability
Highway	Fair Hex Trafficability
Oil Pipeline	Poor Hex Trafficability
Water Pipeline	Very Poor Hex Trafficability
FEBA	Coast
Country Border	City

Table 6. Theater Map Customizable Objects

Neutral Territory	Airbase
Red Territory	Red Ground Unit
Blue Territory	Blue Ground Unit
Ocean	Stacking Dot
Hex Outline	Location Box

convention dictates that the end user should be allowed to decide the colors for the various objects. The application programmer typically specifies default colors which the user can override. One way the user can do this is through entries in the *.Xdefaults* file in the user's root directory.

The user may customize the colors for the main hexboard objects listed in Table 5. Similarly, Table 6 lists the objects that can be customized on the theater map (mini hexboard).

4.2.3.8 Animation. The animation portion Saber user interface allows the game players to see how the previous day's battle unfolded. They can see how their mission orders were carried out in addition to the enemy's response.

The animation process begins with the game player specifying the starting day and time. The hexboard is erased and the map is redrawn to reflect the location and status of the forces at the specified day and time. The battle then commences. Land units may be seen moving from hex to hex as they carry out their orders. Aircraft packages appear on the screen as they are formed. Depending on their mission, the aircraft packages may move across the map in an effort to reach their target. The aircraft packages are removed from the screen at the conclusion of their missions or sooner if the package is destroyed by the enemy.

When an airbase, land unit, or aircraft package is attacked, the hex where they are located momentarily changes color. This enables the game players to visually see any trends or patterns of attack. For example, the enemy may send numerous strike missions against a group of friendly land units to weaken the line of defense in preparation for a major offensive. This type of information may be helpful in planning future missions.

The user has the same controls over the display when animation is active as he did before the animation started. This includes changing the visible terrain features and units, viewing or printing reports, and panning across the display. In addition, the user may bring up the status of any friendly unit. The status information that is displayed changes as necessary when the unit receives or inflicts damage.

Once the animation has started, the user is provided additional controls to pause, resume or quit the animation. If the user quits the animation, the map display reverts back to what it looked like before the animation was begun. If the animation stops because all events were replayed, then the display should already be in the same state as it was before the animation was started.

The database flat files produced at the end of the simulation provide a snapshot of the units and the end of the game day or cycle. This information is useful for a static display of the battlefield. However, the function of the animation routines is to show how the units arrived at their current condition and location. To provide this capability, information is needed about the events which took place between the end of one day and the end of the next.

A history file was developed to fill this information gap. The simulation creates this file by recording information about all significant events in the order in which the events occurred. The entries in the history file file contain all the necessary information to show when and where units moved or were attacked as well as the amount of attrition they suffered.

Deciding what events are significant enough to be included in the history file depends on what is to be displayed and how it is to be represented. For the Saber wargame, an event is significant if:

- it causes a change of system or object status and if the game players need to be aware of the new status, or
- it is needed to calculate statistical summaries for the report generator.

Frequently, a significant event will involve two objects. An example is when unit A attacks unit B. This event could be reported as "Unit A Attacks Unit B," "Unit B Was Attacked By Unit A," or both ways. Reporting an event twice places an unwanted and unnecessary burden on the simulation. Therefore, only one entry is made for such events. The question remains as to which of the first two methods is better. The object-oriented paradigm suggests that a system be defined as objects and the operations to be performed on the objects. Thus, the events are designed with objects being the recipient or target of some action. The example above would then be recorded as "Unit B Was Attacked By Unit A."

The entries in the history file are divided into event records and status records. Each event record contains such information as the time, the asset identification of the major object, the type of the event, and the hex where the action occurred. Some event records may have additional information specific to the type of event.

Event records are followed by zero, one or more status records. The format and content of the status records depend on the type of event record. When the status of a unit changes, the only things that must be written to the history file are the new values for the changed attributes. However, this approach has a significant drawback. Namely, the

simulation would have to perform numerous checks to determine exactly which attributes changed values in order to write them to the history file. To relieve the simulation of this processing, the status records contain fields for the values of all required attributes. The simulation is expected to output a value for each field, even if the values are unchanged. This may result in a greater number of output commands being executed in the simulation, but no checks are needed to determine which values changed. A complete description of the history file is provided in Appendix B.

The overall algorithm for the animation feature is as follows. When the user requests to start animation, a new window is opened and the user is prompted for the requested starting day and time. The default values for these fields are the beginning of the previous day. The end of day information for the day before the requested start day is retrieved into new instances of the appropriate objects and the hexboard is redrawn. For example, if the current day is day 3, and the user requests to see a replay of day 2, then new land unit objects will be created to hold the status of the land units that were saved at the end of day 1.

If the requested start time is other than the beginning of the day, then the hexboard cannot be redrawn right away. Instead, the entries in the history file are read and processed until entries for the requested start time are encountered. No screen updates are issued until entries with the requested start time are found.

The entries in the history file are then sequentially read. Each event record is decoded to determine the object and the action to be performed on the object. Depending on the event type, the status records may also need to be processed. Processing new status values involves a call to the appropriate status update method for the object. Processing of some events requires that the screen be updated. For example, movement events require that the widgets for the unit be erased (unmanaged) at the present location and redrawn (managed) at the destination hex. If a unit is attacked, the background color of the hex where the attack took place is temporarilly changed.

The processing of history file events continues until the end of file is reached, the user pauses the animation, or the user quits the animation.

4.2.3.9 Summary Reports. The detailed design of the report generator for the Saber wargame was driven by the requirements specified in section 4.1.1.2. One of the requirements for the wargame stated that the user must have the capability to view and print reports at any time. However, because of the file input and output that takes place, report creation is a relatively slow process. This led to the following design considerations:

- Regardless of the number of times the user views or prints a report, each report should be created only once.
- The user should not have to wait for reports to be created when viewing or printing reports.

The solution to these issues rested on the fact that the Saber wargame is non-interactive. After analysis of the current situation, the game participants enter their mission input assignments for the next game day (or portion thereof). The players then go away while the simulation is run. When the participants return, the process is repeated.

All of the information needed to generate the Saber reports is available at the end of the simulation run. The required data is either in the database flat files or in the history file produced for the animation portion of the game. Thus, there is no reason for the user interface itself to create the reports.

Instead, a report preprocessor is run at the end of the simulation to produce all of the Saber reports. The report preprocessor outputs each report into a Unix file suitable for printing. If the participants have specified a subset of reports to be printed each day, then the report preprocessor sends the appropriate files to the printer.

The Saber user interface allows the user to view or print any of the previously produced reports by selecting the *Reports* pulldown menu from the main menu bar. If the player wants to print a report, the user interface queues the file for printing as is done in the report preprocessor.

Viewing reports also involves the use of the preprocessor generated report files. OSF Motif allows for files to be read into text widgets and drawing area widgets for viewing and/or modification. The ability to modify the file contents can easily be removed from

the widgets. The text and drawing area widgets also support scroll bars if the file does not fit inside of the window.

4.3 Summary

This chapter presented the system requirements for the Saber wargame. The requirements were gathered primarily through the use of system prototypes and group discussions. The system requirements were followed by a high level design and presentation of how the Saber application fits in with the Ada bindings to the X Window System. Lastly, detailed design issues were presented for the major objects and features. The next chapter presents the implementation of the wargame.

V. Saber Wargame Implementation

This chapter describes the process of implementing the Saber user interface. It begins with a discussion of the usefulness of the various Ada bindings. This is followed by the pros and cons of using the Motif User Interface Language. The chapter ends with a discussion of the specific implementation decisions made during the course of the project.

5.1 Ada Bindings

Three sets of Ada bindings were used to interface with the X Window System and the Air Force Wargaming Center's (AFWC) hex widget. This section first describes the implementation of the Ada bindings to the hex widget. It then describes the benefits and drawbacks of using the Ada bindings written by the Boeing Corporation and the Science Applications International Corporation (SAIC).

5.1.1 Hex Widget Bindings. In order to use the AFWC hex widget, a set of Ada bindings had to be developed. The Ada bindings were modeled after the bindings Boeing developed to the Motif widget set. Each procedure exported by the hex widget had to have a corresponding Ada procedure. To aid in understanding, the Ada procedure names were given the same names as their C counterparts except that underscores were inserted between words. Thus, the C procedure "Hx_SetHexLabel" became "Hx_Set_Hex_Label". The complete binding for this procedure is shown in Figure 25.

As can be seen from the figure, the Ada procedure was implemented with another procedure nested inside of it. The outer procedure is the one called by the application program. Thus, the application program should declare variables of the appropriate type to pass into the procedure. The inner procedure is what is actually bound to the corresponding C procedure. In order to distinguish it to the compiler, it is given the same name as the outer procedure except that all underscores are removed. It should be noted that the inner procedure has no body in the Ada code. Its body is actually the C procedure.

The actual binding was accomplished using the Ada *pragma interface* and *pragma interface_name* constructs. In the figure, the *pragma interface* construct indicates that the

```
procedure HX_Set_Hex_Label( Hex_Widget : in WIDGET;
                            Hex_X      : in AFS_LARGE_NATURAL;
                            Hex_Y      : in AFS_LARGE_NATURAL;
                            Label      : in STRING;
                            Redraw     : in BOOLEAN) is

   procedure HXSetHexLabel( Hex_Widget : in SYSTEM.ADDRESS;
                            Hex_X      : in AFS_LARGE_NATURAL;
                            Hex_Y      : in AFS_LARGE_NATURAL;
                            Label      : in SYSTEM.ADDRESS;
                            Redraw     : in AFS_LARGE_NATURAL );

   pragma INTERFACE (C, HXSetHexLabel);
   pragma INTERFACE_NAME (HXSetHexLabel, "_HX_SetHexLabel");

   Temp_Label : constant STRING := Label & ASCII.NUL;

begin

   HXSetHexLabel( Widget_To_Addr( Hex_Widget ),
                  Hex_X,
                  Hex_Y,
                  Temp_Label(1)'address,
                  BOOLEAN'pos( Redraw ) );

end HX_Set_Hex_Label;
```

Figure 25. Ada Binding to Hx_SetHexLabel

Table 7. Parameter Conversion Rules

Outer Procedure Parameter Type	Mode	Inner Procedure Parameter Type	Method of Type Conversion
Widget	in	System.Address	XT.Widget_To_Addr(variable_name)[a]
	out		XT.Addr_To_Widget(variable_name)
AFS_Large_Natural[b] (integer ≥ 0)	in	AFS_Large_Natural	none
	out	System.Address	variable_name'address
String	in	System.Address	variable_name(1)'address
	out		
Boolean	in	AFS_Large_Natural	Boolean'pos(variable_name)
	out	System.Address	local_variable_name'address

[a]XT is an abbreviation of the Boeing package "X_TOOLKIT_INTRINSICS_OSF"
[b]This type is defined in the Boeing package "AFS_BASIC_TYPES"

inner procedure is to be bound to a procedure written in the language C. The name of the Ada procedure is then paired with the name of the corresponding C procedure through the *pragma interface_name* construct.

The primary purpose of the body of the outer procedure is to convert the Ada input parameters to the types needed by the inner procedure for transfer to the C subroutine. However, the challenge in developing the bindings was determining exactly what types of parameters should be passed to the C procedures. Table 7 was developed to assist in this determination for some of the major data types. Given the type and mode of the parameter in the outer procedure, the table lists the type for the variable in the inner procedure. It also shows how the type conversion should be accomplished in the body of the outer procedure.

In general, if a variable in the outer procedure has a mode of "out", then the corresponding variable in the inner procedure must be of type "System.Address". This is because the C procedure must have the address of the variable if it is going to set or change the value. One other important point is illustrated in Figure 25. In C, all strings must be terminated by an ASCII null character. Ada strings, however, typically do not end with this character. Thus, before sending the string address to the C subroutine, the Ada bindings append an ASCII null.

5.1.2 Boeing Bindings. The bindings written for the Xt Intrinsics and Motif widget set proved to be an indispensable part of the Saber user interface. While there were some weaknesses noted in the software, as a whole the Boeing bindings were able to directly or indirectly satisfy the requirements for the user interface.

5.1.2.1 Weaknesses. The Boeing bindings are not totally perfect. The first problem one notices when looking at the software is the lack of documentation. For the most part, the only documentation is in the form of section titles which separate the subroutines into topical categories. Thus, it would help if the application programmer is already familiar with the Xt Intrinsics and Motif widget set before trying to use the Boeing bindings. Furthermore, a few of the subroutines do not have nice, clean bindings to their corresponding C routines. These Ada subroutines use sparsely documented data structures that are defined within the bindings and that have no counterpart in the C code. It takes some time to learn what these data structures are for and how to use them properly.

The second weakness is that the bindings do not cover every Motif and Xt Intrinsics function. This fact is made clear in a "README" file that comes with the software. Some of the "missing" procedures can be added without too much difficulty. For example, it approximately took 15 minutes to add the *Xt_Error* procedure. Other functions require a little more thought. An example is the *Xt_CloseDisplay* procedure. The straightforward implementation for this procedure was unsuccessful. However, a solution was found by developing a *Xt_Quit_X_Request* exception that is raised when the user wants to quit the application. A modification was then made to the *Xt_Main_Loop* to break out of the infinite loop when this exception is raised.

The third drawback to using the Boeing bindings is that they are currently tied to the Verdix Ada Development System (VADS) version 5.5 or higher. The bindings make use of the "C_Strings", "A_Strings", and "Command_Line" packages provided with the VADS library. The use of these packages restricts the portability of the application software. The "README" file included with the Boeing bindings indicates which modules would have to be changed to port the software to machines with different Ada compilers. However, the required changes should not be attempted by the novice Ada programmer.

5.1.2.2 Hardware Dependencies. Even if your system does have VADS version 5.5 or higher, there is no guarantee that the Boeing bindings will work correctly. This fact was determined the hard way when attempting to use the bindings on a Sun 386i machine running VADS version 5.7 with Unix. In order to gain familiarity with Motif and the Boeing bindings, a simple test program was written. The program created a pushbutton widget and registered a callback procedure to change the text string on the pushbutton when it was pressed. However, the program aborted with a "Segmentation Fault" when it was executed. Analysis of the code showed that it was syntactically and semantically correct.

A week later, it was determined that there were two problems, neither of which were caused by the Boeing bindings or the test program. The causes of the problems were found in the August, 1991 edition of the *VADS Connection.* According to the Verdix Corporation, there are three potential problems areas to be aware of when writing programs that interface with C. These are parameter passing conventions, register usage, and parallelism. In this case, it was the first two areas that were causing the test program to abort.

The Verdix Corporation described the parameter passing conventions as follows[59:8]:

> In many cases, C does not use the same parameter passing conventions as Ada. When calling C from Ada this is not a problem, because VADS automatically generates a C calling sequence whenever pragma INTERFACE is used. When calling Ada from C, however, there can be a problem. Verdix has implemented pragma EXTERNAL, which will cause an Ada subprogram to accept a C calling sequence, but this is only available in version 6.0.5 and above.

The problem encountered with register usage had to do with differences in the ways Ada and C use registers. According to the Verdix Corporation[59:8]:

> For the 386...C expects the call to save and restore any registers it modifies, other than eax. Ada expects the caller to do the saving. This works fine when Ada calls C, but screws things up when C calls Ada. These register saves must be done manually, through the use of machine_code insertions.

At first glance, it did not appear that these issues would be causing the problems. It was obvious that Ada was making calls to C through the Boeing bindings, but it was not readily apparent that C was making any calls back to Ada. However, C was making calls to Ada inside of the *Xt_Main_Loop* procedure. Specifically, after the pushbutton is pressed, the C procedure *Xt_DispatchEvent* eventually causes control to be passed back to the Ada callback procedure that was registered with the pushbutton. It was at this point that the abovementioned problems caused the "Segmentation Fault".

However, it should be stressed that this was not a problem with the Boeing bindings. Rather, it is inherent in the way callback procedures are dispatched. The test program and the Boeing bindings worked correctly when the software was executed on a Sun Sparc Station 2.

5.1.2.3 Strengths. The weaknesses of the Boeing bindings are outweighed by its strengths. For example, the bindings were found to be very well written. No errors or problems were found with any of the Boeing written subprograms used in the Saber user interface. Furthermore, while there were a few exceptions, most of the Ada subprograms bear a close resemblance to their C counterparts. Thus, anyone familiar with the calling sequences for the Xt Intrinsics and the Motif widget set should be able to understand the functionality of Ada programs that use the Boeing bindings.

5.1.3 SAIC Bindings. The SAIC bindings were needed for the creation of the pixmaps used in the Saber user interface. Pixmaps were used to display the symbols for the airbases, ground units, and aircraft missions. The SAIC code provided bindings to Xlib routines for reading the bitmap data from files. Then the bindings were used to convert the data into the pixmap data structure needed to make the symbol appear on the pushbutton widgets. With one exception, the SAIC bindings performed correctly.

The problem involved reading the bitmap data from the files. The data files were created by the *BITMAP* editor provided with the X Window System software. This simple drawing program allows an application programmer to interactively create bitmap patterns.

The pattern is saved in a special format that can be read in by an application program through calls to appropriate Xlib subroutines.

The *BITMAP* program outputs the bitmap data in groups of two hexadecimal digits. Thus, each of theses two digit numbers is in the range 0 . . FF (or, in decimal, 0 . . 255). However, the SAIC bindings read each two digit number into an eight bit data structure called "Bit_Data" that can only handle numbers in the range $-2^7 . . 2^7 - 1$ (or, $-128 . . 127$). This means that any hexadecimal number greater than 7F is considered out of range.

Two solutions to this problem were considered. The first, which was deemed impractical, involved changing the hexadecimal data produced by *BITMAP* to the decimal equivalents, while maintaining the same binary representation. To make the decimal numbers fit in the proper range, each decimal number greater than 127 must be subtracted by 256. For example,

$$\begin{aligned} &\textit{old number}: \quad F8_{16} && = 248_{10} = 11111000_2 \\ &\textit{new number}: \quad 248_{10} - 256_{10} && = -8_{10} = 11111000_2 \end{aligned}$$

The solution that was implemented, however, was much easier. According to the documentation, the SAIC programmers made a previous attempt to correct this problem. In doing so, they changed the range of permissible values and constrained the size of the data structure to only eight bits. The solution to the problem, then, involved changing the range of permissible values back to 0 . . 255. Testing showed that this change solved the problem without creating new ones.

5.1.4 Combining the Boeing and SAIC Bindings. As was previously mentioned, both the Boeing bindings and the SAIC bindings were used in the development of the Saber user interface. The Boeing bindings were the primary means of interfacing with the X Window System, while the SAIC bindings were used primarily for the creation of the pixmaps for the unit symbols. Making the few calls to the SAIC bindings was not straightforward because of inconsistent types used by the two sets of bindings. Some

inconsistencies were resolved by simple type conversion while others required the addition of new subroutines to the software.

5.1.4.1 Type Conversions. By necessity, the Boeing software contains Ada declarations of a few low-level Xlib routines. These declarations for such things as the X Window System display, windows, and drawables were needed because the Xt Intrinsics provides functions to return these values that are created when the connection with the X server is established and windows are displayed on the screen.

Several of the SAIC procedures used to create the unit symbol pixmaps required these values as parameters. Two methods were used to convert the values to the types needed by the SAIC code. The first was a simple type conversion as in the following example that converts a float number to an integer:

```
integer_number := integer( float_number );
```

The second method used unchecked conversion, a predefined generic function provided as part of the Ada language. This generic function had to be instantiated with a source type and a target type for each conversion to be performed. An example instantiation to convert a variable of type "Display_Pointer" returned by Boeing's *Xt_Display* function to a variable of type "Display" for use in the SAIC routines follows:

```
function Display_Id_From_Xt_Display is new Unchecked_Conversion
                                ( Source => XLIB.Display_Pointer,
                                  Target => X_Windows.Display );
```

The unchecked conversion utility allows a sequence of bits, an address in the above example, to be treated as a variable of two different types. However, this capability should be used with caution. As Cohen writes, "Abuse of this capability can subvert the elaborate consistency-checking mechanisms built into the Ada language and lead to improper internal representations for data"[11:804]. For the Saber user interface, however, this was the only way to pass certain variables created through the Boeing bindings as input parameters to the SAIC subroutines.

5.1.4.2 Problems With SAIC Data Structures. Since the initial connection with the X server was made through the Xt Intrinsics via the Boeing bindings, and not through the SAIC code, several internal SAIC data structures were not initialized. Because these data structures were not initialized, some functions provided by the SAIC bindings could not be used.

Two of the functions that fell into this category were *Default_Depth* and *Root_Window*. The results returned by these functions were needed for the creation of the unit symbol pixmaps. To obtain these values, a binding was created for each function and added to the Boeing bindings. Before the values could be used by the SAIC subroutines, however, they had to be converted to the corresponding SAIC types. The value returned by *Root_Window* was converted using the unchecked conversion described in the previous section, while the value returned by *Default_Depth* was converted through simple type conversion.

5.2 Using the Motif User Interface Language

The Open Software Foundation (OSF) provides a User Interface Language (UIL) and Motif Resource Manager (MRM) as part of the basic OSF/Motif environment. The UIL is a specification language for describing the appearance and behavior of Motif objects for a user interface. Through the UIL, the applications programmer can specify the widgets to be used and the callback functions to be entered when the user interface changes state as a result of user actions.

To use the interface specifications, the UIL file must first be compiled to produce a User Interface Definition (UID) file. Typically, compilation of the UIL file takes much less time than compilation of the application file. In the application program, MRM functions are called to initialize the connection with the UID file. Additional MRM functions are used to access the interface definitions and cause the Motif objects to appear on the screen.

Following is a discussion of the advantages and disadvantages of using the UIL and MRM to create a user interface.

5.2.1 Advantages of UIL and MRM. The Open Software Foundation lists the following benefits of creating a Motif based user interface using UIL and MRM[40:III-1-2]:

- Easier Coding. You can specify an interface faster using UIL because you do not have to know specific widget creation functions or the format of their argument lists.
- Earlier Error Detection. The UIL compiler does the type checking for you that is not available with the Motif or X Toolkits, so that the interface you specify has fewer errors.
- Separation of Form and Function. When you use UIL, you define your application interface in a separate UIL module rather than by directly calling Motif Toolkit creation functions in your application program.
- Faster Prototype Development. You can create a variety of interfaces in a fairly short time, and get an idea of the look of each interface before the functional routines are written.
- Interface Customization. You can customize an interface by putting in place a hierarchy of UID files (called a UID hierarchy). At run time, MRM searches this file hierarchy in the sequence you specify to build the appropriate argument lists for the widget creation functions.

Another advantage is the capability to fine tune the layout of the user interface. For example, UIL makes it easy to adjust the placement and margins of various pushbuttons inside of a bulletin board widget. Because recompilation of the UIL file is so fast, various combinations can be tried until the interface meets the programmer's satisfaction.

5.2.2 Drawbacks. There are several significant disadvantages to using UIL and MRM in the implementation of a user interface. These include portability problems, the overhead of numerous procedure calls, and problems with implementing an object-oriented design.

The first, and perhaps most significant, drawback is that OSF does not guarantee that the UIL and MRM functions will be upward compatible with future releases of Motif. While the *OSF/Motif Programmer's Guide* praises UIL and MRM, another OSF publication does not.

OSF has published a four volume collection of books referred to as the Application Environment Specification (AES). The AES was designed to list the specifications for

programming user interfaces, such that applications using only the included items would be guaranteed to operate consistently on a wide variety of hardware platforms[39]. In order to be included in the AES, the interface elements must be stable (i.e., not likely to change) and reliable.

Unfortunately, the UIL and MRM were excluded from the Motif AES. The User Environment volume of the AES lists the following rationale for its exclusion[39:1-16]:

> Preliminary feedback from builders of interactive design tools and user interface management systems indicates that as is, uil (sic) does not completely support their needs. While we hope that future changes will be upward compatible, this cannot be guaranteed at this time.

This is not an idle threat. During the development of the Saber user interface, two UIL items were found to be incompatible between Motif releases 1.0 and 1.1. The first involved the callback reason *XmNcreateCallback*. With release 1.1, this callback was no longer supported. The name of the callback had changed to *MrmNcreateCallback*. The second problem with UIL involved the scrolled window widget. The UIL component of release 1.0 generated horizontal and vertical scroll bars by default. However, the scroll bars must be explicitly requested with the later version.

The second disadvantage to using UIL and MRM deals with the extra overhead involved in numerous procedure calls. To begin with, several calls are needed just to establish a connection with the UID file. Once the connection is established, many more subroutine calls are needed whenever the application program needs to know the widget ids of UIL created widgets. Since the widgets are created through the UIL and not in the application program, the widget ids must be passed to the application software by specifying a *MrmNcreateCallback* for each needed widget. Thus, for a bulletin board widget with twelve toggle buttons, twelve separate callbacks are needed whenever the bulletin board is created. Once the application program has the twelve widget ids, another twelve procedure calls are needed to set the initial state of the toggle buttons.

A third disadvantage of using UIL and MRM relates to the implementation of an object-oriented design. For the Saber user interface, several bulletin board widgets with

multiple toggle buttons were needed. Using UIL, the exact physical layout and behavior of each bulletin board must be specified in advance. This includes specifying such things as the main title, instructions, names for each toggle button, and callback procedures.

Since we have several bulletin board widgets with similar properties, the object-oriented approach suggests creating a "Toggle Button Board" class that is instantiated whenever a list of user customizable settings is to be displayed. Objects are instantiated with the specific titles, instructions, callbacks, and a list of toggle buttons. The initial settings for the toggle buttons can be specified in advance, before the buttons are created. Also, the widget ids are saved as each toggle button is created in the application program, thus reducing the number of procedure calls, while at the same time, resulting in more of an object-oriented implementation. Unfortunately, the UIL approach does not support this paradigm.

5.2.3 Suggested Uses of the UIL. Even with its disadvantages, the UIL can, and should, still be used as part of user interface development. Specifically, the UIL is a good tool for rapid prototyping, interface design, specifying widget hierarchies, and fine tuning of user interface objects. However, the UIL specifications should be translated into the appropriate Motif subroutine calls so that the widgets are created in the application program. The application program will then be more portable and more efficient. At the same time, it will be a better implementation of an object-oriented design.

5.3 User Interface Implementation

The following sections describe some of the more significant issued faced during the implementation of the Saber user interface. First, a general description is presented of the packages used to implement the objects and controlling routines. Since a description of the objects is given in Chapter 4 and Appendix A, this section focuses on the controlling packages. This description is followed by a more detailed summary of the animation controller package. Lastly, a description is given of changes that were made to the Air Force Wargaming Center's hex widget.

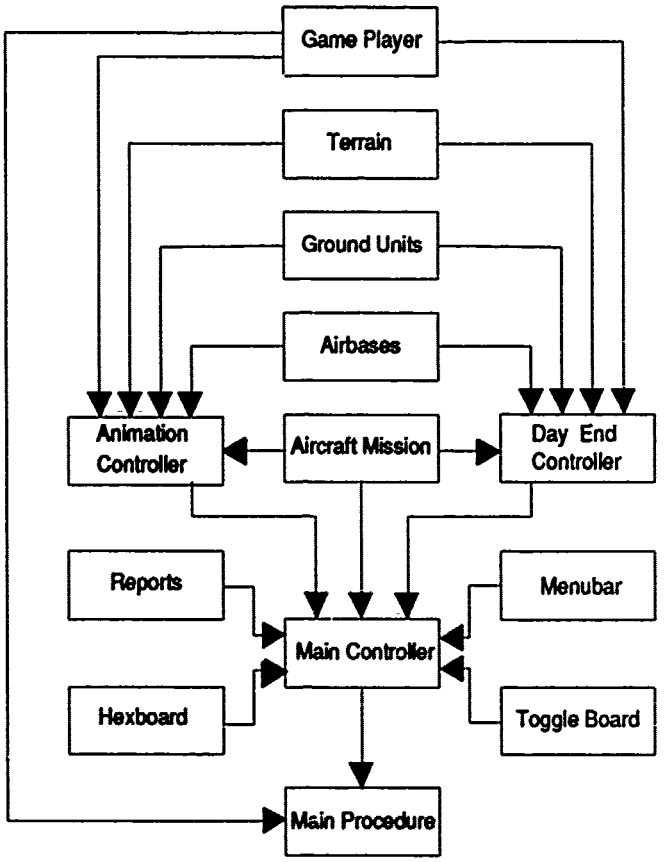


Figure 26. Saber Module Diagram

5.3.1 Ada Package Implementation. Figure 26 shows the Ada package implementation of the Saber user interface using a variant of Booch's method[7]. In this figure, packages at the heads of the arrows depend on the packages at the tails of the arrows. In other words, a package at the head of an arrow must "WITH" the package at the tail. A package was developed for each object class shown in Figure 15. Each package contains subroutines for the creation and manipulation of individual instances of the object.

The controlling functions of the Saber user interface are handled in the main procedure and in three controlling packages. The Main_Procedure of Figure 26 creates the start up form then enters the *Xt_Main_Loop* to process the X events. The three controller packages instantiate the objects and contain the callback procedures for all menu and button press events. The Main_Controller instantiates objects that are shared by the Day_End_

Controller and the Animation_Controller. These objects include the toggle button boards, menubars, and hexboards. These objects will exist whether animation is active or the user is viewing the end of day unit positions.

The objects instantiated by the subordinate controllers are mutually exclusive of each other. While objects may be instantiated in both controllers at the same time, only one controller will have control over the hexboards at a time. In other words, only one controller will be displaying its objects on the screen at a time. The objects instantiated by the subordinate controllers include terrain features, ground units, airbases, and aircraft packages. The Day_End_Controller instantiates static objects that represent the state of the simulation at the end of a day's action. The Animation_Controller, on the other hand, instantiates objects from a previous day. The attributes of the Animation_Controller's objects change as the animation is run. It should be noted that the day end instantiations are held unchanged even if animation is started. This makes it faster to redisplay the end of day positions if the animation is aborted before reaching the end of the history file.

5.3.2 Animation_Controller. This subcontroller performs all actions associated with recreating the battle engagements of previous days. It allows the game players to visually see the results of their employment decisions. The information needed to perform the animation is acquired from both the database flat files and the history file. The database flat files are used to instantiate objects reflecting the status of the battlefield at the start of the animation period. The entries in the history file are then read in sequential order to update the status and location of the forces.

The requirement that most significantly impacted the implementation of the Animation_Controller was that the user should have the capability to perform all the same functions that they could before the animation was started. Once begun, the main portion of the animation process consists of entering a loop to read and process all of the events in the history file. However, if the user is to be able to perform other functions while the animation is active, some means was necessary to occasionally stop reading the history file in order to process the user requests. To accomodate this requirement, the Animation_Controller was implemented as an Ada task inside of a package. Instead of procedures, the

task has entry points that are called by the Main_Controller. The general outline of the task is shown in Figure 27.

The task does nothing until the user requests to start the animation by clicking on the appropriate pulldown menu item on the main menubar. To begin the animation, the Main_Controller calls the Start_Animation entry point to establish an Ada rendezvous. During this rendezvous, the initial terrain, ground unit and airbase objects are instantiated and displayed on the screen.

After creation of the objects, the task enters a loop where entry calls are accepted and history file processing takes place. If there are no waiting entry calls and the animation is not paused, then an event from the history file is read and processed. Otherwise, the waiting entry call is accepted and a rendezvous takes place. If more than one entry call is waiting, the Ada language dictates that one will be arbitrarily selected for acceptance.

The entry calls that will be accepted are shown in Figure 27. Two of the entry points are for redisplaying terrain and units after the user indicated that the visibility of certain objects should be altered. Another two entry points handle requests to display or remove weather overlays and hex status windows.

The Fill_In_Theater_Map entry adds the graphical representations of the objects to the theater map. The Stop_Animation entry will cause a permanent halt of the animation. The history file is closed and the user is presented with a small window that allows him to reset the display to the end of day positions. The Pause_Animation entry causes a temporary halt of the animation. The history file remains open, but unread, until the animation is either permanently halted or resumed.

When the animation is active, there is always the chance that the Animation_Controller task and the main program may both try to execute an Xlib, Xt Intrinsics or Motif function at the same time or shortly after one another. If the first call gets blocked before the function completes, the second call could arbitrarily manipulate data structures or memory locations still needed by the first call.

One way to prevent this from happening is to use a monitor to protect the data structures from concurrent update by two or more tasks. For this project, however, the

```
loop
   select
      accept Start_Animation
         <set User_Wants_To_Continue to TRUE>

      while User_Wants_To_Continue loop
         select
            accept Redisplay_Terrain
         or
            accept Redisplay_Units
         or
            accept Weather_Display_Toggle_Changed
         or
            accept Show_Hex_Info
         or
            accept Fill_In_Theater_Map
         or
            accept Suspend_Window_Operations
            accept Resume_Window_Operations
         or
            accept Pause_Animation
               <set User_Is_Pausing to TRUE>
         or
            when User_Is_Pausing =>
                accept Resume_Animation
                   <set User_Is_Pausing to FALSE>
         or
            accept Stop_Animation
               <set User_Wants_To_Continue to FALSE>
               <set User_Is_Pausing to FALSE>
         else
            if not User_Is_Pausing and not End_Of_History_File then
                <Read And Process History File>
            end if
         end select
      end loop
   or Terminate
   end select
end loop
```

Figure 27. Outline of Animation Controller Task

use of monitors was not realistic because of the numerous functions and data structures provided by the X Window System.

Instead, an entry was added to the Animation_Controller to momemtarily suspend the animation process whenever the Main_Controller needs to execute an X Window System function. Suspending the Animation_Controller prevents it from executing a simultaneous request. The Main_Controller then allows the Animation_Controller to resume processing after the X Window System function completes execution. Because of the nature of the Ada *accept* statements, if the Animation_Controller is itself performing window operations when the Main_Controller requests the suspension, the Main_Controller will have to wait until the window operations are completed. At that time the Animation_Controller will accept the call. It is important to note that the Day_End_Controller has no need to synchronize with the Animation_Controller in this manner because its procedures will not be executed while the animation is active.

Another significant issue with the Animation_Controller concerned how the task entries should be called. Some entries are only called by the Main_Controller. For others, the most efficient way is to have the task entry act like a callback procedure. To do so, the address of the entry must be specified when registering callbacks. Unfortunately, Ada will not allow the *address* attribute to be applied to task entry names directly. A solution to this problem was to introduce a level of indirection by making all callbacks go to the Main_Controller which can then make a normal task entry call.

5.3.3 Changes to the Hex Widget. The hex widget developed by the Air Force Wargaming Center (AFWC) provided the means for drawing the hexboard and terrain features. For the most part, the hex widget is well written and easy to use. Furthermore, the addition of new routines for drawing railroads, pipelines, and various hexside assets was fairly straightforward. Two changes were made to the hex widget to correct and error in erasing assets and to decrease the time it takes to redraw the hexboard.

The hex widget treats each hex as an individual object and keeps the terrain features for each hex in its own internal data structures. These internal data structures are used to actually display the terrain features on the hexboard. Terrain features are added to

the hexes by repetitive calls to the appropriate hex widget procedures. These procedures create a graphics context for each added feature. A graphics context is an Xlib data structure that specifies such things as the foreground and background colors to be used when drawing objects on the screen. When adding radial or hexside assets to a hex, a separate call must be made for each side of the hex that is to receive the asset. Thus, if a road enters a hex from the north side and exits to the south side, two calls must be made to the hex widget. One call draws a road from the center of the hex to the north side, and the other from the center to the south side. The hexboard is not drawn on the screen, or "realized", until all terrain assets have been added to the hexboard.

To remove a terrain feature from a hex, the hex widget's internal data structure must be changed. Removal of a city from a hex is accomplished by adding a city with a radius of zero. This overwrites the old city value. Similarly, radial and hexside assets are removed by adding the particular feature with a width of zero. Because the Day_End_Controller maintains the original set of terrain features, the asset data is not lost. The terrain features can be easily restored to the hexes.

The first attempt at removing a terrain feature revealed an error in the hex widget. A road segment was removed from a hex by setting its width to zero. However, when the hex was redrawn, there was still a road segment one pixel wide. The problem was with the test performed by the hex widget to determine if the road segment should be drawn. The hex widget checked to see if a graphics context existed. Since the graphics context was never deleted, the road segment was redrawn.

A simple fix to the problem would have been to delete the graphics context when adding a road with a width of zero. However, doing so would mean that the graphics context would have to be recreated if the user chose to display roads again. Thus, the solution was not implemented in this manner. Instead, the test to determine which assets should be drawn was changed to check to see if the road segment had a width of zero.

The other change made to the AFWC's hex widget reduced the time it takes to redraw the terrain features on the hexboard. As terrain objects are added to the hexboard, the hex widget checks to see if the hexboard has been realized, or displayed on the screen. If it

has not, as is the case when initially creating the hexboard, the hexes are not drawn until after the last terrain feature has been added. However, if the hexboard is currently being displayed on the screen, then a hex is redrawn every time a terrain feature is added to or removed from it. Thus, if a hex has a three road segments in it and the user has requested that roads be removed from the hexboard, then the hex will be redrawn three times; once after removing each segment.

To eliminate this unnecessary redrawing of the hexes, a parameter was added to each procedure that adds features to hexes. The parameter is a boolean flag that indicates whether the hex should be redrawn after the terrain asset is added to the hex.

5.4 Summary

This chapter described the significant issues of the Saber user interface implementation. A description was given of how bindings were developed for the AFWC hex widget. This was followed by the benefits and drawbacks of the Boeing and SAIC bindings. While the bindings generally performed well, some modifications were necessary to correct minor deficiencies. Next, the pros and cons of the Motif User Interface Language (UIL) were given. The UIL can be used as an effective prototyping tool. However, because of upgrade problems, the UIL should not be used as part of the final user interface. The chapter ended with a description of the packages used to implement the object-oriented design. The implementation of the animation routines as entry points in an Ada task was described in some detail.

The next chapter summarizes the entire thesis effort and gives recommendations for future expansions to the user interface.

VI. Conclusions and Recommendations

This chapter summarizes the work performed for this thesis. It then presents some recommendations for further work to be performed on the user interface.

6.1 Summary

This thesis effort resulted in the development of an animated graphical user interface for the Saber wargame. It provides a post-game look at the status of forces and allows the game player to graphically see how the battle has progressed through the movement of unit icons around the battlefield. The work to generate this project was accomplished using an iterative approach. The major actions performed were:

- Analysis of the problem domain. Before doing any design or implementation, an intensive study was accomplished to gain an understanding of the terms, concepts, and philosophies needed to design a user interface for a wargame. This involved research in the areas of wargames, user interface design, and the X Window System.
- Determination of system requirements. A series of screen and report prototypes were developed to gain a better understanding of the requirements for the user interface. Other requirements were gathered through group discussions and through examination of similar wargames in various stages of development.
- Development of the high-level design. Having laid the foundation through the accomplishment of the previous two steps, the next task was the high-level, object-oriented design of the user interface. This involved the identification of the objects, their attributes, and the communication needed between them. At this point, several alternatives of how to incorporate the X Window System into the design were considered. The decision was made to access the Xlib routines through the SAIC produced bindings and access the Xt Intrinsics and Motif widget set using the bindings produced by Boeing.
- Iterative generation of code. The software was generated through a repetitive process of evaluation and planning, detailed design, code generation, and unit and regression

testing. This method ensured that a working product was available at the end of each iteration.

The efforts just described resulted in a user interface that is both portable and extensible. By using the X Window System to develop the user interface, the software is guaranteed to execute on a number of hardware platforms. However, because of the use of the Boeing bindings, the development of the system is restricted to systems using certain versions of the Verdix Ada Development System (VADS). The use of the Motif widget set provides a standard look and feel so that users familiar with other applications using the same widget set will have less of a problem becoming productive on the Saber user interface. The use of the object-oriented paradigm should also make it easy to expand the capabilities of the interface. The functionality of each object is encapsulated into individual procedures to isolate the impact of changes to the system.

6.2 Recommendations

This development effort proved that the Ada programming language can be used to generate graphical user interfaces for wargames. While the Saber user interface is a good beginning, there were certain features that could not be implemented due to time constraints. Also, the implementation process revealed other features and capabilities that can be achieved using the Motif widget set. Thus, the following ideas are provided as areas of potential enhancement:

- *Three dimensional representation of the battlefield.* The X11R5 release of the X Window System includes PEX, a PHIGS/+ extension to X that allows for the generation of three dimensional images. While some work may be necessary to interface the Ada language to this new release, it would open up many more possibilities for the user interface.
- *Improved intelligence reporting.* The intelligence portion of the user interface could be improved through the use of shadow records to provide the status and location of enemy forces based on the latest intelligence data. The shadow records would be used to report information to the user while its actual information would be stored

elsewhere. As more intelligence is gathered for a particular unit, the information in the shadow record would be updated. Another improvement would be to provide predictions of enemy action based on the gathered intelligence data.

- *The use of the Motif paned window widget.* The Motif widget set provides a paned window widget which allows multiple widgets to be placed in vertical "panes". An optional "sash" can be used to separate the child widgets. The user can adjust the size of each individual pane using a control box on the sash. There are a couple of potential uses for the paned widget in the Saber user interface. One use would be to display different portions of the map in different panes. Another use would be to show the end of day positions in one pane and run the animation in another.

- *The use of gadgets instead of widgets.* Creating widgets causes new windows to be generated on the display. Each window uses up both application and X Window server resources. Also, creating numerous windows tends to slow down an application. To alleviate these problems, Motif provides things called gadgets which are basically windowless widgets. Gadgets are available for many of the commonly used widgets such as push buttons, toggle buttons, and separators. They perform many of the same functions as their widget counterparts, but consume fewer resources and can be created faster. According to Young, gadgets cannot support event handlers, translations, or popup children, although they do support callback functions[63]. Thus, an area of improvement would be to determine where the widgets could be replaced with gadgets in the Saber user interface.

- *Development of file naming conventions for the database files.* The animation portion of the Saber user interface requires the capability to read the database flat files for any previous day. Thus, some naming convention is necessary to distinguish the files for day one from the files for day two, three, etc.

- *Expansion of "help" features.* The Saber user interface was developed using single page help screens. This capability should be expanded to provide multiple levels of help for each topic.

- *Addition of "hot keys" for the experienced user.* The user interface presently allows the game player to step through the menu items using the keyboard or mouse. Mnemonics are provided that allow the user to easily select a menu item without stepping down to a particular choice. However, many experienced users may prefer not to use the menu system at all. The Motif widget set offers a solution to this problem by allowing "hot keys" to be set up for the menu items. A hot key is a programmer specified key combination such as "Alt-A" or "Control-Q" that can be used to perform a specific function without going through the various menus. The Saber user interface should be expanded to provide this capability.
- *Design and implementation of a scenario development tool.* This tool should provide the wargame staff the capability to develop a wargame for any part of the world. It should allow for the initial placement of ground units, and airbases anywhere on the globe. It should also allow for the addition of supplies and equipment to any of the air or ground units.

6.3 Conclusions

In conclusion, this thesis documented the design, rationale, and implementation of the Saber user interface. It showed how the Ada programming language could be successfully used to develop an object-oriented user interface using the X Window System, the Xt Intrinsics and the Motif widget set. This work forms the baseline for future efforts at completing an integrated Ada wargame that can help teach air and ground employment doctrine to the future leaders of the United States Air Force and its allies.

Appendix A. *Saber Class Descriptions*

This appendix describes the application and Motif classes which make up the Saber user interface. The classes are designed to be generic enough to be implemented in any programming language. The attributes and methods are described for each class.

A.1 Application Classes

A.1.1 Game Player Class. This class contains information about the person playing the game. The methods collect and return this information.

A.1.1.1 Attribute Types

Seminar_Number_Type: string for the seminar number of the class

Current_Day_Type: integer representing the current day relative to the start of the battle

Player_Side_Type: indicates whether the player is on the RED, BLUE, or WHITE team

A.1.1.2 Methods. These are the methods for the Game Player class. The purpose, input parameters, and output parameters are given for each method.

Fetch_Start_Up_Form

- Purpose: generates the start up form to retrieve player information
- Inputs: addresses of "CONTINUE" and "QUIT" callback procedures
- Outputs: widget id of the start up form

Is_Red

- Purpose: function that returns true if player side is RED
- Inputs: none
- Outputs: none

Is_Blue

- Purpose: function that returns true if player side is BLUE
- Inputs: none
- Outputs: none

Is_White

- Purpose: function that returns true if player side is WHITE
- Inputs: none
- Outputs: none

Get_Seminar_Number

- Purpose: function that returns the seminar number
- Inputs: none
- Outputs: none

Get_Current_Day

- Purpose: function that returns the current day
- Inputs: none
- Outputs: none

Set_User_Values

- Purpose: procedure that extracts and saves the values for the Seminar Number, Current Day, and Player side from the widgets.
- Inputs: none
- Outputs: none

A.1.2 Terrain Class. Objects of this class contain information about the terrain covering the hexboard. The methods read and display the terrain information.

A.1.2.1 Attribute Types

Hex_Range: Variables of this type are integers between 0 and 99.

Neighbor_Range: Variables of this type are integers between 1 and 9999.

Hex_Array_Type: Variables of this type are arrays of asset information for the hexes. The asset information includes center hex id, sides of the hex that form part of air hex borders, force, country, weather zone, weather, intel index, combat power in, combat power out, terrain type, amount of forestation, pie trafficablility for each hex side, hex sides containing roads, railroads, and pipelines.

Neighbor_Array_Type: Variables of this type are arrays, indexed by neighbor id, that contain the two hexes and their hex sides that are neighbors.

River_Segment_Pointer: This is a pointer to a list of neighbor id's and river sizes that form the rivers.

Obstacle_Pointer: This is a pointer to a list of hexside obstacles. The following information is kept for each obstacle: obstacle id, neighbor id, name, and visibility to BLUE and RED forces.

City_Pointer: This is a pointer to a list of cities. The following information is kept for each city: name, hex location, size, population, and whether or not it is a capital.

Neighbor_List_Pointer: Variables of this type are pointers to a linked list of neighbor id's. This type is used for FEBA and country border lists.

Terrain_Object_Type: This is the type for the Terrain object and is instantiated through a call to *Read_Terrain_Data*. Variables of this type contain all terrain information about a hexboard. This type consists of a consolidation (i.e., record or structure) of the following variables listed with their corresponding types in parenthesis:

Hex_Array (Hex_Array_Type)	Border_List (Neighbor_List_Pointer)
Neighbor_Array (Neighbor_Array_Type)	FEBA_List (Neighbor_List_Pointer)
Obstacle_List (Obstacle_List_Pointer)	City_List (City_Pointer)
River_Segment_List (River_Segment_Pointer)	Max_X (Hex_Range)
Max_Neighbor (Neighbor_Range)	Max_Y (Hex_Range)

A.1.2.2 Methods. This section lists the methods for the Terrain class. The purpose, input parameters, and output parameters are given for each method.

Read_Terrain_Data

- Purpose: This procedure creates instances of variables of type Terrain_Object_Type by reading the specified database flat files.
- Inputs: Filenames for the Hex, Travel, City, Hexside Assets, FEBA, Roads, Railroads, and Pipelines database relations.
- Outputs: A variable of type Terrain_Object_Type instantiated with data from the input files.

Show_Terrain_Data

- Purpose: This procedure displays all of the terrain data (except the weather) on the given hexboard subject to the current terrain display toggle buttons.
- Inputs: Hexboard Widget, Terrain Object, Terrain Display Toggle Button List
- Outputs: none

Show_Weather_Data

- Purpose: This procedure displays the current weather on the given hexboard.
- Inputs: Hexboard Widget, Terrain Object
- Outputs: none

Erase_Weather_Data

- Purpose: This procedure removes the current weather from the given hexboard.
- Inputs: Hexboard Widget, Terrain Object
- Outputs: none

Show_Minefield

- Purpose: This procedure adds a minefield to a specific side of a hex on the given hexboard. The hex is redrawn if the user is currently displaying minefields.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y Coordinates, Hex Side, Terrain Display Toggle Button List
- Outputs: none

Erase_Minefield

- Purpose: This procedure removes a single minefield from the given hexboard and the Terrain data structure.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates, Hex Side
- Outputs: none

Show_Bridge

- Purpose: This procedure adds a bridge to specific sides of two hexes on the given hexboard. The hex is redrawn if the user is currently displaying bridges.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates for each hex, Hex Sides, Terrain Display Toggle Button List
- Outputs: none

Erase_Bridge

- Purpose: This procedure removes a single bridge from the given hexboard and the Terrain data structure.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates for each hex, Hex Sides
- Outputs: none

Flash_Hex_Background

- Purpose: This procedure rapidly changes the background color of a user specified hexagon to indicate the hex or units in the hex have been attacked.
- Inputs: Hex X and Y coordinates
- Outputs: none

A.1.3 Hexboard Class. Objects of this class are hexboards in which terrain and/or units can be displayed. The methods create and manipulate the visibil portion of the hexboard.

A.1.3.1 Attribute Types

Game_Board_Type: This is the type for the Game Board (hexboard) object and is instantiated through a call to *Create_Hexboard.* Variables of this type contain the widget id, width, height of the hexboard as well as the current X, Y, W, H locations. Additionally, they contain the graphics context for the location box and an indication of weather or not the first exposure of the hexboard has occurred.

Hexboard_Client_Data_Type: This is the type for the client data to be passed to hexboard callback procedures. It consists of Game_Board_Type instantiations for the main hexboard and the theater map. It also contains the widget id of the scrolled window widget containing the main hexboard.

A.1.3.2 Methods. This section lists the methods for the Hexboard class. The purpose, input parameters, and output parameters are given for each method.

Create_Hexboard

- Purpose: This function creates a hexboard of the specified size and with the specified options.
- Inputs: Parent, Name, Background Color, Hex Outline Color, Stacking Dot Color, number of hexes in the X and Y directions, Hex Radius, and whether or not to display Hex Labels
- Outputs: an instantiated object of type Game_Board_Type

Set_GC_Location_Box

- Purpose: This procedure sets the graphics context for the location box. The location box is drawn in the theater map and shows the portion of the theater that is being displayed in the main hexboard
- Inputs: Theater Map, X Windows Display, X Windows Drawable, Location Box Color
- Outputs: none

Theater_Map_Button_Press

- Purpose: This procedure adjusts the visible portion of the main hexboard as a result of the user pressing the mouse button while the sprite is inside the theater map.
- Inputs: Hexboard Client Data
- Outputs: none

Draw_Location_Box

- Purpose: This procedure draws the location box in the theater map.
- Inputs: Hexboard Client Data
- Outputs: none

Set_Theater_Map_Active

- Purpose: This procedure sets a flag to indicate that the theater map is being displayed.
- Inputs: none
- Outputs: none

Theater_Map_Is_Active

- Purpose: This function returns "true" if the theater map is being displayed. Otherwise, it returns "false".
- Inputs: none
- Outputs: boolean value indicating if theater map is currently being displayed

A.1.4 Ground Unit Class. Objects of this class contain information about the ground units covering the hexboard. The methods read and display the ground units and their status.

A.1.4.1 Attribute Types

Ground_Unit_Pointer: This is a pointer to a list of ground units. The following information is kept for each ground unit:

- Static Information: unit designator, unit type, country, force, corps id, supported units, supporting units
- Status Information: mission, target number, combat power, firepower, surface-to-air index, total ammunition, total hardware, total petroleum-oil-lubricants (POL), amount of water, amount of engineer support, intel index, visible to enemy indicator, weather, hex location
- Widget Information: widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.4.2 Methods. This section lists the methods for the Ground Unit class. The purpose, input parameters, and output parameters are given for each method.

Set_GC_Ground_Symbols

- Purpose: This procedure sets the graphics context for the ground unit symbols.
- Inputs: X Windows Display, X Windows Drawable, Red Foreground and Background Colors, Blue Foreground and Background Colors
- Outputs: none

Initialize_Ground_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue ground unit symbols.
- Inputs: Display Id
- Outputs: none

Read_Ground_Unit_Data

- Purpose: This procedure creates instances of type Ground_Unit_Pointer by reading the specified database flat files.
- Inputs: Filenames for the Land Unit, Unit Supports, Move, Move LNLT, Unit Components, and Unit G2A database relations.
- Outputs: Two variables of type Ground_Unit_Pointer that point to a list of ground units. One pointer is returned for the Red units and one for the Blue units.

Get_Ground_Unit

- Purpose: This function returns a pointer to a specific ground unit's information.
- Inputs: Ground Unit List, Unit Id
- Outputs: pointer to the ground unit with the specified Unit Id

Show_All_Ground_Units

- Purpose: This procedure displays all the ground units (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit List for a particular side (force)
- Outputs: none

Erase_All_Ground_Units

- Purpose: This procedure erases all the ground units (for a particular side) from the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit List for a particular side (force)
- Outputs: none

Erase_Single_Ground_Unit

- Purpose: This procedure erases a single ground unit from the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit Pointer for a particular unit
- Outputs: none

Move_Ground_Unit

- Purpose: This procedure moves a single ground unit on the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with new location information

Show_Unit_Status

- Purpose: This procedure opens a status window for a particular ground unit.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with the widget id of the status window added

Erase_Unit_Status

- Purpose: This procedure closes a status window for a particular ground unit.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with the widget id of the status window removed

Update_Unit_Status

- Purpose: This procedure updates the status for a particular ground unit. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit, new Status Information
- Outputs: the Ground Unit Pointer with updated Status Information

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular unit has a status window open.
- Inputs: Ground Unit Pointer for a particular unit
- Outputs: "true" if the ground unit has an open status window, "false" otherwise

A.1.5 Aircraft Mission Class. Objects of this class contain information about the aircraft missions covering the hexboard. The methods display the aircraft missions and their status.

A.1.5.1 Attribute Types

Aircraft_Mission_Pointer: This is a pointer to a list of aircraft missions. The following information is kept for each aircraft mission:

- Static Information: aircraft package number, mission, target, requested time on target, altitude, hex level
- Aircraft Information: hex location, the starting number, current number, and types of aircraft flying as Primary, Suppression of Enemy Air Defense (SEAD), Electronic Counter Measures (ECM), Refueling, and Escort aircraft.
- Widget Information: widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.5.2 Methods. This section lists the methods for the Aircraft Mission class. The purpose, input parameters, and output parameters are given for each method.

Set_GC_Aircraft_Symbols

- Purpose: This procedure sets the graphics context for the aircraft mission symbols.
- Inputs: X Windows Display, X Windows Drawable, Red Foreground and Background Colors, Blue Foreground and Background Colors
- Outputs: none

Initialize_Aircraft_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue aircraft mission symbols.
- Inputs: Display Id
- Outputs: none

Add_Aircraft_Mission

- Purpose: This procedure adds an instance of type Aircraft_Mission_Pointer to the Aircraft Mission List.
- Inputs: Static Information, Aircraft Information, Aircraft Mission List, Main Hexboard, Theater Map, an indication to draw the mission on the game boards.
- Outputs: Aircraft Mission List with the new mission added

Get_Aircraft_Mission

- Purpose: This function returns a pointer to a specific aircraft mission's information.
- Inputs: Aircraft Mission List, Mission Id
- Outputs: pointer to the aircraft mission with the specified Mission Id

Show_All_Aircraft_Missions

- Purpose: This procedure displays all the aircraft missions (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission List for a particular side (force)
- Outputs: none

Erase_All_Aircraft_Missions

- Purpose: This procedure erases all the aircraft missions (for a particular side) from the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission List for a particular side (force)
- Outputs: none

Erase_Single_Aircraft_Mission

- Purpose: This procedure erases a single aircraft mission from the map. It also removes the mission from the list of aircraft missions.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission Pointer for a particular unit
- Outputs: none

Move_Aircraft_Mission

- Purpose: This procedure moves a single aircraft mission on the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission Pointer for a particular unit an indication to draw the mission on the game boards.
- Outputs: the Aircraft Mission Pointer with new location information

Show_Mission_Status

- Purpose: This procedure opens a status window for a particular aircraft mission.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit
- Outputs: the Aircraft Mission Pointer with the widget id of the status window added

Erase_Mission_Status

- Purpose: This procedure closes a status window for a particular aircraft mission.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit
- Outputs: the Aircraft Mission Pointer with the widget id of the status window removed

Update_Mission_Status

- Purpose: This procedure updates the status for a particular aircraft mission. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit, new Status Information
- Outputs: the Aircraft Mission Pointer with updated Status Information

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular unit has a status window open.
- Inputs: Aircraft Mission Pointer for a particular unit
- Outputs: "true" if the aircraft mission has an open status window, "false" otherwise

A.1.6 Airbase Class. Objects of this class contain information about the aircraft bases covering the hexboard. The methods read and display the aircraft bases and their status.

A.1.6.1 Attribute Types

Airbase_Pointer: This is a pointer to a list of aircraft bases. The following information is kept for each aircraft base:

- Static Information: base id, name, command, country, force, higher headquarters, total POL storage, maximum ramp space
- Status Information: ramp space available, alternate fields, intel index, number of enemy mines, status, MOPP posture, POL on base, POL in hard storage, maintenance personnel on hand, maintenance hours accumulated, amount of maintenance equipment, amount of spare parts, number of shelters, number of EOD crews, number of rapid runway repair (RRR) crews, an indicator of the base's visibility to the enemy, and the weather
- Widget Information: widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.6.2 Methods. This section lists the methods for the Airbase class. The purpose, input parameters, and output parameters are given for each method.

Initialize_Airbase_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue airbase symbols.
- Inputs: Display Id, Red foreground and background colors, Blue foreground and background colors.
- Outputs: none

Read_Airbase_Data

- Purpose: This procedure creates instances of type Airbase_Pointer by reading the specified database flat files.
- Inputs: Filenames for the Airbase, Depot, Runways, Alternate Bases, Airbase Aircraft, Airbase Weapons, and Weather database relations
- Outputs: Two variables of type Airbase_Pointer that point to a list of airbases. One pointer is returned for the Red bases and one for the Blue bases.

Get_Airbase

- Purpose: This function returns a pointer to a specific aircraft base's information.
- Inputs: Airbase List, Base Id
- Outputs: pointer to the aircraft base with the specified Base Id

Show_All_Airbases

- Purpose: This procedure displays all the aircraft bases (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Airbase List for a particular side (force)
- Outputs: none

Erase_All_Airbases

- Purpose: This procedure erases all the aircraft bases (for a particular side) from the map.
- Inputs: Main Hexboard, Theater Map, Airbase List for a particular side (force)
- Outputs: none

Erase_Single_Airbase

- Purpose: This procedure erases a single aircraft base from the map. It also removes the base from the list of aircraft bases.
- Inputs: Main Hexboard, Theater Map, Airbase Pointer for a particular unit
- Outputs: none

Show_Airbase_Status

- Purpose: This procedure opens a status window for a particular aircraft base.
- Inputs: Main Hexboard, Airbase Pointer for a particular unit
- Outputs: the Aircraft Base Pointer with the widget id of the status window added

Erase_Airbase_Status

- Purpose: This procedure closes a status window for a particular aircraft base.
- Inputs: Main Hexboard, Airbase Pointer for a particular unit
- Outputs: the Aircraft Base Pointer with the widget id of the status window removed

Update_Airbase_Status

- Purpose: This procedure updates the status for a particular aircraft base. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Airbase Pointer for a particular unit, new Status Information
- Outputs: the Aircraft Base Pointer with updated Status Information

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular airbase has a status window open.
- Inputs: Airbase Pointer for a particular unit
- Outputs: "true" if the aircraft base has an open status window, "false" otherwise

A.1.7 Report Class. Objects of this class contain status information about ground units, aircraft missions, and airbases. The methods display and print the various reports.

A.1.7.1 Attribute Types

Report_Pointer: This is a pointer to a list of report names.

A.1.7.2 Methods. This section lists the methods for the Report class. The purpose, input parameters, and output parameters are given for each method.

Display_Options_Menu

- Purpose: This procedure displays the main report options menu.
- Inputs: none
- Outputs: none

View_Report

- Purpose: This procedure opens a window in which a report is displayed.
- Inputs: Report Name to display
- Outputs: none

Print_Report

- Purpose: This function sends a list of reports to a printer.
- Inputs: Report List
- Outputs: none

Select_Standard_Set

- Purpose: This function allows the user to select a standard set of reports for daily printing.
- Inputs: none
- Outputs: Report List

A.2 Motif Classes

A.2.1 Toggle Button Board Class. Objects of this class are bulletin board widgets that contain various toggle buttons for various custom settings. The methods create and manipulate the toggle button boards. print the various reports.

A.2.1.1 Attribute Types

Button_Record: This is a collection of information about a toggle button. It includes such information as: the button name, widget id, current value, new value, and a value changed callback address.

Button_List: This is a pointer to a list of button records.

A.2.1.2 Methods. This section lists the methods for the Toggle Button Board class. The purpose, input parameters, and output parameters are given for each method.

Make_Button_List

- Purpose: This procedure creates an empty Button List.
- Inputs: none
- Outputs: pointer to an empty Button List

Clear_Button_List

- Purpose: This procedure empties a Button List so it can be reused.
- Inputs: Button List
- Outputs: Button List

Set_Button

- Purpose: This procedure adds information for a new button to the specified Button List
- Inputs: Button List, Button Record information
- Outputs: Button List with new Button Record added

Create_Toggle_Button_Board

- Purpose: This procedure creates a bulletin board for a set of toggle buttons.
- Inputs: Button List, Board Title, Instructions, OK Callback Address, CANCEL Callback Address, HELP Callback address
- Outputs: none

Toggle_Button_Value_Changed

- Purpose: This procedure handles *XmNvalueChanged* callbacks. It records the new value of a toggle button.
- Inputs: Button List
- Outputs: Button List with new value

Reset_Toggle_Values

- Purpose: This procedure resets the values for the toggle buttons back to the state they were in before the toggle button bulletin board was created.
- Inputs: Button List
- Outputs: Button List with values reset to their initial state

Set_New_Toggle_Values

- Purpose: This procedure sets the new values for the toggle buttons to the state they were in when the user closed the toggle button bulletin board.
- Inputs: Button List
- Outputs: Button List with values set to their new state.

A.2.2 Menubar Class. Objects of this class are menubar widgets that contain various pulldown menus. The methods create the menubar and allow for the addition of pulldown menus and the items on the pulldown menus.

A.2.2.1 Attribute Types

none

A.2.2.2 Methods. This section lists the methods for the Menubar class. The purpose, input parameters, and output parameters are given for each method.

Create_Menubar

- Purpose: This function creates a menubar with a help pulldown menu on the far right side. Other pulldown menus can be added through the *Create_Pulldown_Menu* procedure.
- Inputs: Parent Widget, Help Message
- Outputs: widget id of menubar

Create_Pulldown_Menu

- Purpose: This function creates a cascade button on the specified menubar and a pulldown menu to hang off of it.
- Inputs: Parent Menubar Widget, Name On Menubar, Mnemonic, Pulldown Title
- Outputs: widget id of the pulldown menu

Add_Pulldown_Menu_Item

- Purpose: This function adds a menu item to a pulldown menu. A pushbutton is created for the menu item with the specified callback.
- Inputs: Pulldown Menu Widget, Item Name, Callback Address
- Outputs: widget id of the newly created pushbutton

Appendix B. *Saber History File*

This appendix describes the entries for the history file. The history file serves two purposes for the postprocessor. First, it is used in the generation of certain reports where the information cannot be obtained directly from the files written out by the simulation. The second function is to provide a script to be used in the animation of the day's action. The entries in the history file contain the necessary information to show when and where units moved as well as the amount of attrition they suffered..

To reduce the number of entries in the history file, only events which cause a change of system (or object) status are recorded. Additionally, whenever possible, each event is only recorded once. Thus, instead of recording that unit A attacked unit B and that unit B was attacked by unit A, only one entry will be made. Each event is action oriented in that an object performs some action or is the target of some action.

The entries in the history file are divided into event records and status records. The event records the object performed the action, the time the event occurred, and the type of event. Depending on the type of the event, it may be followed by one or more status records. These records reflect the new status of the object. The format of the status records also depends on the type of the event.

The following sections describe the format and contents of the event and status records.

B.1 Events Affecting Aircraft Package Status

List of Events:

MS1) MSNSTRT = Mission Start
MS2) MOVE = Move
MS3) ATKDBY = Attacked By
MS4) JETTSN = Jettison
MS5) MSNCOMP = Mission Complete

B.1.1 MS1 - Mission Start

B.1.1.1 Event Record

```
                    Rendez              Msn  Rqst
E Time Asset_Id Hex_Id    Event    Type TOT  Force Target

Example:
E 1630 MS000016 HX030225 MSNSTRT OCA  1800 BLUE  HX012341
```

B.1.1.2 Status Record: (one for each aircraft type in the aircraft package)

```
  Msn  Acft  Num   Num   Rand PMS  Wx
S Cat  Type  Rqstd Avail Abrt Abrt Abrt

Example:
S PRIM F15D  15    13    1    1    1
S ESC  F16E  5     5     0    0    0
S REF  KC135 1     1     0    0    0
```

NOTE: The MSNSTRT status records should be followed by Aircraft Departed (ACDEP) event and status records for each airbase that contributed to the aircraft package.

B.1.2 MS2 - Move

B.1.2.1 Event Record

```
E Time Asset_Id New_Hex  Event

Example:
E 1645 MS000016 HX030245 MOVE
```

B.1.3 MS3 - Attacked By

B.1.3.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Attacker

Example:
E 1700 MS000016 HX030245 ATKDBY  MS000144
```

NOTE: The attacker field could be an aircraft package or a ground unit.

B.1.3.2 Status Record:

```
  Msn  Acft  New
S Cat  Type  Num

Example:
S PRIM F15D  11
S ESC  F16E  4
S REF  KC135 1
```

B.1.4 MS4 - Jettison

B.1.4.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1730 MS000016 HX030245 JETTSN
```

B.1.4.2 Status Record: (one for each aircraft type in the aircraft package)

```
  Msn  Acft  New
S Cat  Type  Num

Example:
S PRIM F15D  10
S ESC  F16E  0
S REF  KC135 0
```

B.1.5 MS5 - Mission Complete

B.1.5.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1800 MS000016 HX030217 MSNCOMP
```

NOTE: The MSNCOMP event record should be followed by Aircraft Arrived (ACARR) event and status records for each airbase that is to receive returning aircraft.

B.2 Events Affecting Airbase Status

List of Events:

AB1) ATKDBY = Attacked By
AB2) ACDEP = Aircraft Depart
AB3) ACARR = Aircraft Arrive (returning missions or from staging base)
AB4) SUPARR = Supplies Arrive (from depot)
AB5) INTELD = Was Intelled (via tactical reconnaissance mission

B.2.1 AB1 - Attacked By

B.2.1.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Attacker

Example:
E 1630 AB000143 HX010219 ATKDBY  MS000017
```

B.2.1.2 Status Record #1 - Supplies:

```
            Intl Ramp Enmy MOPP POL  POL  Main Main Main Spar      EOD  RRR
S1 Status   Indx Aval Mine Post Soft Hard Pers Hour Eqpt Part Shlt Crew Crew

Example:
S1 ACTIVE  29.2 7500 0    0    2500 1500 150  1750 5000 5000 25   2    2
```

B.2.1.3 Status Record #2 - Aircraft:

```
   Acft  New  Acft  New  Acft  New  Acft  New  Acft  New  Acft  New
S2 Type  Numb Type  Numb Type  Numb Type  Numb Type  Numb Type  Numb

Example:
S2 F15D  32   F16E  22   KC135 7
```

B.2.1.4 Status Record #3 - Weapons:

```
                New                New                New                New
S3 Weapon_Type  Numb Weapon_Type   Numb Weapon_Type   Numb Weapon_Type   Numb

Example:
S3 AIM9L        155  AIM7          552  GBU1000       1000
```

B.2.2 AB2 - Aircraft Depart

B.2.2.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1730 AB000143 HX010219 ACDEP
```

B.2.2.2 Status Records. Uses the status records S1, S2, S3 for Supplies, Aircraft, and Weapons.

B.2.3 AB3 - Aircraft Arrive

B.2.3.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1730 AB000143 HX010219 ACARR
```

B.2.3.2 Status Records. Uses the status records S1, S2, S3 for Supplies, Aircraft, and Weapons.

B.2.4 AB4 - Supplies Arrive

B.2.4.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1730 AB000143 HX010219 SUPARR
```

B.2.4.2 Status Records. Uses the status records S1, S2, S3 for Supplies, Aircraft, and Weapons. All three status records may not be required in every case.

B.2.5 AB5 - Was Intelled

B.2.5.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Recce_Pkg Intel_Index

Example:
E 0900 AB000143 HX010219 INTELD  MS000141  43.0
```

B.3 Events Affecting Depot Status

List of Events:

DP1) ATKDBY = Attacked By
DP2) SUPDEP = Supplies Depart
DP3) INTELD = Was Intelled (via tactical reconnaissance mission

B.3.1 DP1 - Attacked By

B.3.1.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Attacker
```

Example:

```
E 1700 DP000828 HX010201 ATKDBY  MS000017
```

B.3.1.2 Status Record

```
          Intl MOPP POL  POL  Spar      EOD
S Status  Indx Post Soft Hard Part Shlt Crew
```

Example:

```
S ACTIVE  9.2  0    2500 1500 5000 25   2
```

B.3.2 DP2 - Supplies Depart

B.3.2.1 Event Record

```
E Time Asset_Id Hex_Id   Event
```

Example:

```
E 1700 DP000828 HX010201 SUPDEP
```

B.3.2.2 Status Record

```
          Intl MOPP POL  POL  Spar      EOD
S Status  Indx Post Soft Hard Part Shlt Crew

Example:
S ACTIVE  9.2  0    2200 1300 4000 25   1
```

Note: A SUPDEP event should be followed by a SUPARR for an airbase or ground unit.

B.3.3 DP3 - Was Intelled

B.3.3.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Recce_Pkg Intel_Index

Example:
E 0900 DP000828 HX010201 INTELD  MS000141  43.0
```

B.4 Events Affecting Ground Unit Status

List of Events:

GR1) MOVE = Move
GR2) ATKDBY = Attacked By
GR3) SUPARR = Supplies Arrived
GR4) NEWMSN = New Mission
GR5) INTELD = Was Intelled (via tactical reconnaissance mission

B.4.1 GR1 - Move

B.4.1.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1700 LU000443 HX010203 MOVE
```

B.4.2 GR2 – Attacked By

B.4.2.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Attacker

Example:
E 1730 LU000443 HX010203 ATKDBY  LU001044
```

B.4.2.2 Status Record

```
  Intl Intl Cbt  Fire  1   2   3   4   5  Total Total Total Amt  Amt
S Indx Fltr Pwr  Pwr  SAI SAI SAI SAI SAI Ammo  Hrdwr POL   H20  Eng

Example:
S 29.2 1.0  14.0 24.0 0   4   4   10  0   2500  3000  2150  1000 250
```

B.4.3 GR3 – Supplies Arrive

B.4.3.1 Event Record

```
E Time Asset_Id Hex_Id   Event

Example:
E 1800 LU000443 HX010203 SUPARR
```

B.4.3.2 Status Record

```
  Intl Intl Cbt  Fire  1   2   3   4   5  Total Total Total Amt  Amt
S Indx Fltr Pwr  Pwr  SAI SAI SAI SAI SAI Ammo  Hrdwr POL   H20  Eng

Example:
S 29.2 1.0  14.0 24.0 0   4   4   10  0   3000  3290  2300  1250 250
```

B.4.4 GR4 - New Mission

B.4.4.1 Event Record

```
E Time Asset_Id Hex_Id   Event  New_Msn Target

Example:
E 1830 LU000443 HX010203 NEWMSN DEF     HX010203
```

B.4.5 GR5 - Was Inteled

B.4.5.1 Event Record

```
E Time Asset_Id Hex_Id   Event  Recce_Pkg Intel_Index

Example:
E 0900 LU000443 HX010203 INTELD MS000141  43.0
```

B.5 Events Affecting Satellite Status

List of Events:

ST1) LAUNCH = Satellite Launch
ST2) ATKDBY = Attacked By
ST3) MOVE = Move

B.5.1 ST1 - Satellite Launch

B.5.1.1 Event Record

```
E Time Asset_Id Start_Hx Event   Sat_Type  Orbit_Type

Example:
E 1700 ST002319 HX014525 LAUNCH  PHOTO_REC GEOSYNCH
```

B.5.1.2 Status Record

```
S Tgt_Hex  Tgt_Hex  Tgt_Hex  Tgt_Hex  Tgt_Hex  Tgt_Hex  Tgt_Hex
```

Example:

```
S HX014523 HX014524 HX014525 HX014526 HX014527 HX014528
```

B.5.2 ST2 - Attacked By

B.5.2.1 Event Record

```
E Time Asset_Id Hex_Id   Event   Attacker New_Status
```

Example:

```
E 1900 ST002319 HX074523 ATKDBY  LU000199 INACTIVE
```

B.5.3 ST3 - Move

B.5.3.1 Event Record

```
E Time Asset_Id New_Hex  Event
```

Example:

```
E 2100 ST002319 HX074525 MOVE
```

B.5.3.2 Status Record

```
  Intl Intl Cbt  Fire  1   2   3   4   5  Total Total Total Amt  Amt
S Indx Fltr Pwr  Pwr  SAI SAI SAI SAI SAI Ammo  Hrdwr POL   H20  Eng
```

Example:

```
S 29.2 1.0  14.0 24.0 0   4   4   10  0   2500  3000  2150  1000 250
```

B.6 Events Affecting Supply Trains

List of Events:

LG1) LGSTRT = Supply Train Start
LG2) MOVE = Move
LG3) ATKDBY = Attacked By
LG4) LGCOMP = Supply Train Complete

B.6.1 LG1 - Supply Train Start

B.6.1.1 Event Record

```
                                                    Transit
E Time Asset_Id Start_Hx Event  Mode     NumVeh Time

Example:
E 1000 LG006001 HX010321 LGSTRT TRUCK   30     8
```

B.6.1.2 Status Record

```
S Type   Amt    Type   Amt    Type   Amt    Type   Amt

Example:
S SPARES 40     POL    32
```

Note: The LGSTRT event should be followed by a SUPDEP event from a depot.

B.6.2 LG2 - Move

B.6.2.1 Event Record

```
E Time Asset_Id New_Hex  Event

Example:
E 1100 LG006001 HX010322 MOVE
```

B.6.3 LG3 - Attacked By

B.6.3.1 Event Record

```
E Time Asset_Id Hex_Id   Event  Attacker

Example:
E 1115 LG006001 HX010322 ATKDBY MS000143
```

B.6.3.2 Status Record

```
S NumVeh Type   Amt    Type   Amt    Type   Amt    Type   Amt

Example:
S 27     SPARES 32     POL    19
```

B.6.4 LG4 - Supply Train Complete

B.6.4.1 Event Record

```
E Time Asset_Id New_Hex  Event

Example:
E 2100 LG006001 HX010329 LGCOMP
```

Note: The LGCOMP event should be followed by a SUPARR event for a ground unit or an airbase.

B.7 Events Affecting Hex Status

List of Events:

HX1) ATKDBY = Attacked By
HX2) MINED = Mines Laid
HX3) CLRMIN = Clear Mines
HX4) BRIDGE = New Bridge Built
HX5) BRBLWN = Bridge Blown

B.7.1 HX1 - Attacked By

B.7.1.1 Event Record

```
E Time Asset_Id Event  Attacker

Example:
E 1700 HX010203 ATKDBY MS000143
```

B.7.2 HX2 - Mines Laid

B.7.2.1 Event Record

```
E Time Asset_Id Event  Attacker Hex_Side

Example:
E 1700 HX010307 MINED  MS000143 N
```

B.7.3 HX3 - Clear Mines

B.7.3.1 Event Record

```
E Time Asset_Id Event  Attacker Hex_Side

Example:
E 1700 HX010307 CLRMIN MS000143 N
```

B.7.4 HX4 - New Bridge Built

B.7.4.1 Event Record

```
E Time Asset_Id Event  Builder  Hex_Side

Example:
E 1700 HX010314 BRIDGE LU000521 S
```

B.7.5 HX5 - Bridge Blown

B.7.5.1 Event Record

```
E Time Asset_Id Event  Builder  Hex_Side

Example:
E 1700 HX010314 BRBLWN LU000521 S
```

B.8 Events Affecting the Weather

List of Events:

WX1) WXCHNG = Weather Change

B.8.1 WX1 - Weather Change

B.8.1.1 Event Record

```
E Time Event

Example:
E 1700 WXCHNG
```

B.8.1.2 Status Record

```
S Wx_Zone New_Wx

Example:
S 1       GOOD
S 2       POOR
S 3       FAIR
S 4       FAIR
S 5       POOR
```

B.9 Example Script

```
E 1630 MS000016 HX030225 MSNSTRT OCA  1800 BLUE  HX012341
S PRIM F15D  15    13    1    1    0
S ESC  F16E  5     5     0    0    0
S REF  KC135 1     1     0    0    0
E 1630 AB000143 HX010219 ACDEP
S1 ACTIVE 29.2 7500 0    0    2400 1500 150  1750 4900 4900 25  2   2
S2 F15D  32   KC135 7
S3 AIM9L       155   AIM7        552   GBU1000     1000
E 1630 AB000144 HX010228 ACDEP
S1 ACTIVE 22.1 7300 0    0    2100 1900 120  1900 5000 3000 21  5   7
S2 F16E  21   A10A  45
S3 AIM9L       130   AIM7        4000
E 1645 MS000016 HX030325 MOVE
E 1700 MS000016 HX030329 MOVE
E 1700 MS000016 HX030229 ATKDBY  MS000144
S PRIM F15D  11
S ESC  F16E  4
S REF  KC135 1
E 1730 MS000016 HX030325 MOVE
E 1745 MS000016 HX030225 MOVE
E 1800 MS000016 HX030225 MSNCOMP
E 1800 AB000143 HX010219 ACARR
S1 ACTIVE 29.2 7100 0    0    3000 1500 150  1500 4800 4644 25  2   2
S2 F15D  43   KC135 8
E 1800 AB000144 HX010228 ACARR
S1 ACTIVE 22.1 6999 0    0    2100 1900 120  1643 3751 2791 21  5   7
S2 F16E  25   A10A  45
S3 AIM9L       133   AIM7        4000
```

Bibliography

1. Ada Information Clearinghouse. *Available Ada Bindings*. Draft. Lanham, MD, October 1991.
2. Apple Computer, Inc. *Human Interface Guidelines: The Apple Desktop Interface*. Technical Report. Reading MA: Addison-Wesley, 1987.
3. Battilega, John A. and others. "Overview." In Hughes, Wayne P., editor, *Military Modeling*, Military Operations Research Society, 1984.
4. Biles, William E. and Susan T. Wilson. "Animated Graphics and Computer Simulation." In *Proceedings of the 1987 Winter Simulation Conference*, pages 472–477, The Winter Simulation Conference, Atlanta, 1987.
5. Binnie, Michael J. and David L. Martin. "The Role of Animation in Decision-Making." In *Proceedings of the 1988 Winter Simulation Conference*, pages 272–276, The Winter Simulation Conference, San Diego, 1988.
6. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, pages 61–72 (May 1988).
7. Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings, 1991.
8. Brunner, Daniel T. and others. "A General Purpose Animator." In *Proceedings of the 1989 Winter Simulation Conference*, pages 155–163, The Winter Simulation Conference, Washington D.C., 1989.
9. Cammarata, Stephanie and others. "Dependencies and Graphical Interfaces in Object-Oriented Simulation Languages." In *Proceedings of the 1987 Winter Simulation Conference*, pages 507–517, The Winter Simulation Conference, Atlanta, 1987.
10. Chignell, Mark H. and John A. Waterworth. "WIMPS and NERDS: An Extended View of the User Interface," *SIGCHI Bulletin*, *23*:15–21 (April 1991).
11. Cohen, Norman H. *Ada as a Second Language*. New York: McGraw-Hill, 1986.
12. Department of the Army. *Operational Terms and Symbols*. FM 101-5-1. Washington: HQ USA, 21 October 1985.
13. Dunnigan, James F. *How to Make War*. New York: William Morrow, 1988.
14. Finn, Richard M. *CRES Theater Game Requirements*. Technical Report. Maxwell AFB, Alabama, 1989.
15. Gordon, Peter J. *A Graphical Player Interface to the Theater War Exercise*. MS thesis, AFIT/GCS/ENG/89D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.
16. Grudin, Jonathan. "The Case Against User Interface Consistency," *Communications of the ACM*, *32*:1164–1173 (October 1989).
17. Halloran, Capt Tim J. "Hex Widget." C computer software source code, 1990.

18. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards*, January 1989.

19. Hodgson, Gordon M. and Stephen R. Ruth. "The Use of Menus in the Design of On-Line Systems: A Retrospective View," *SIGCHI Bulletin*, *17*:16–22 (July 1985).

20. Horton, Capt Andre M. *Design and Implementation of a Graphical User Interface and a Database Management System for the Saber Wargame.* MS thesis, AFIT/GCS/ENG/91D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

21. Hyland, Stephen J. and Mark A. Nelson. "Ada Bindings to the X Window System." Ada computer software source code, 1987.

22. *Interface Standards Informal Technical Data, Ada Interfaces to X Window System.* Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1069(NP), Reston VA: Unisys Corporation, March 1989 (AD-A228820).

23. John A. Madden, Lt Col. "Conceptual Design and Development of Joint Service Wargame, Part I." U.S. Army War College, June 1982.

24. Johnson, Eric F. and Kevin Reichard. *X Window Applications Programming.* Portland: MIS Press, 1989.

25. Johnson, Eric F. and Kevin Reichard. *Power Programming ... Motif.* Portland: MIS Press, 1991.

26. Jones, E. J. "Ada Bindings to the Xt Intrinsics and Motif Widget Set." Ada computer software source code, 1991.

27. Jones, Oliver. *Introduction to the X Window System.* Englewood Cliffs NJ: Prentice Hall, 1989.

28. Koivunen, Marja-Ritta and Martii Mantyla. "HutWindows: An Improved Architecture for a User Interface Management System," *IEEE Computer Graphics and Applications*, pages 43–52 (January 1988).

29. Kross, Capt Mark S. *Developing New User Interfaces for the Theater War Exercise.* MS thesis, AFIT/GCS/ENG/87-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A189744).

30. Lowgren, Jonas. "History, State and Future of User Interface Management Systems," *SIGCHI Bulletin*, *20*:32–44 (July 1988).

31. Mann, Capt William F. III. *Saber: A Theater Level Wargame.* MS thesis, AFIT/GOR/ENS/91M-9, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991 (AD-A238825).

32. Myers, Brad A. "A Taxonomy of Window Manager User Interfaces," *IEEE Computer Graphics and Applications*, *8*:65–84 (September 1988).

33. Myers, Brad A. Software Design: User Interface Design (2), Video tape number AC-SD-01-25. Carnegie Mellon University, Software Engineering Institute, 1989.

34. Myers, Brad A. Software Design: User Interface Design (1), Video tape number AC-SD-01-24. Carnegie Mellon University, Software Engineering Institute, 1989.

35. Myers, Brad A. and Mary Beth Rosson. "User Interface Programming Survey," *SIGCHI Bulletin*, *23*:27–30 (April 1991).

36. Ness, Capt Marlin A. *A New Land Battle for the Theater War Exercise*. MS thesis, AFIT/GE/ENG/90J-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1990 (AD-A223087).

37. Nielsen, Jakob. "Coordinating User Interfaces for Consistency," *SIGCHI Bulletin*, pages 63–65 (1989).

38. Nye, Adrian and others. *Xlib Reference Manual for Version 11, Volume One*. Massachusetts: O'Reilly and Associates, Inc, 1988.

39. Open Software Foundation. *Application Environment Specification: User Environment Volume*. Revision A. Englewood Cliffs NJ: Prentice Hall, 1990.

40. Open Software Foundation. *OSF/Motif Programmer's Guide*. Revision 1.0. Englewood Cliffs NJ: Prentice Hall, 1990.

41. Open Software Foundation. *OSF/Motif Style Guide*. Revision 1.0. Englewood Cliffs NJ: Prentice Hall, 1990.

42. Perla, Peter P. *The Art of Wargaming*. Annapolis MD: Naval Institute Press, 1990.

43. Pountain, Dick. "The X Window System," *Byte*, *14*:353–360 (January 1989).

44. Quick, Darrell. *A Graphics Interface for the Theater War Exercise*. MS thesis, AFIT/GCS/ENG/88D-16, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988 (AD-A205902).

45. Ross, R. *Entity Modeling: Techniques and Application*. Boston MA: Database Research Group, 1987.

46. Roth, Mark A. "Personal conversations," (January through November 1991).

47. Scheifler, Robert W. and others. *X Window System: C Library and Protocol Reference*. Digital Press, 1988.

48. Scheifler, Robert W. and Jim Gettys. "The X Window System," *ACM Transactions on Graphics*, *5*:79–109 (April 1986).

49. Schneiderman, Ben. *Designing the User Interface*. Reading MA: Addison-Wesley, 1987.

50. Sheridan, Robert E. "The Script Processing Technique in Modeling/Simulation and its Role in the Generation of Animated Computer Graphics." In *Proceedings of the 1986 Winter Simulation Conference*, pages 810–825, The Winter Simulation Conference, Washington D.C., 1986.

51. Sherry, Capt Christine. *Object-Oriented Analysis and Design of the Saber Wargame*. MS thesis, AFIT/GCS/ENG/91D-21, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

52. Simon, Major Glenn D. *Interactive Graphical Support for a Small-Unit Amphibious Operation Combat Model.* MS thesis, Naval Postgraduate School, Monterey, CA, March 1983 (AD-A128561).

53. Smith, Sydney L. and Jane N. Mosier. *Guidelines for Designing User Interface Software.* Contract F19628-86-C-0001, Bedford MA: MITRE Corporation, August 1986 (AD-A177198).

54. Sommerville, Ian. *Software Engineering.* Massachusetts: Addison-Wesley, 1989.

55. Stevens, Lt Nora G. *The Application of Current User Interface Technology to Interactive Wargaming Systems.* MS thesis, Naval Postgraduate School, Monterey, CA, September 1987 (AD-A186856).

56. Szekely, Pedro. "Separating the User Interface from the Functionality of Application Programs," *SIGCHI Bulletin*, *18*:45–46 (October 1986).

57. Tevis, Jay-Evan J. II. *An Ada-Based Framework for an $IDEF_0$ CASE Tool Using the X Window System.* MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A189681).

58. Trumbly, James E. and Kirk P. Arnett. "Including a User Interface Management System (UIMS) in the Performance Relationship Model," *SIGCHI Bulletin*, *20*:56–62 (April 1989).

59. Verdix Corporation. *VADS Connection.* Technical Report. Chantilly, VA, August 1991.

60. Wallnau, Kurt C. *Ada/Xt Architecture: Design Report.* Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1107(NP), Reston VA: Unisys Corporation, January 1990 (AD-A228827).

61. Wallnau, Kurt C. and others. *Ada/Xt Toolkit, Version Description Document.* Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1133(NP), Reston VA: Unisys Corporation, July 1990 (AD-A229637).

62. Young, Douglas. *X Window Systems: Programming and Applications with Xt.* Englewood Cliffs NJ: Prentice Hall, 1989.

63. Young, Douglas. *The X Window System: Programming and Applications with Xt (OSF/Motif Edition).* Englewood Cliffs NJ: Prentice Hall, 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
AN ANIMATED GRAPHICAL POSTPROCESSOR FOR THE SABER WARGAME

5. FUNDING NUMBERS

6. AUTHOR(S)
Gary W. Klabunde, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-10

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AU CADRE/WG
Maxwell AFB, AL 36112

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

One of the most cost effective ways to learn and hone the skills necessary to conduct and win a war is through the use of realistic computer simulations of conflict, or wargames. The Saber wargame was developed for just this purpose. Saber is a multi-sided, theater-level simulation developed by the Air Force Institute of Technology for the Air Force Wargaming Center. It models conventional, nuclear, and chemical warfare between aggregated air and ground forces. To aid in the realism, the effects of logistics, satellites, weather, and intelligence are represented. Saber provides an avenue for senior level joint service officers to improve their airpower employment decisioni-making skills. This thesis documents the object-oriented design and implementation of the graphical post-processor for the Saber wargame. The user interface provides the game players with the force status information necessary to plan and execute a theater-level air war. The interface includes a report processor that produces reports for on screen viewing or printing. The system also provides animation capabilities to allow the game players to see how the day's battle unfolded in an effort to enhance the learning process. The user interface was written in the Ada programming language using the X Window System and OSF/Motif widget set. Ada bindings developed by the Boeing Aerospace Corporation and the Science Applications International Corporation (SAIC) were used to interface to the various X libraries. These bindings were supplemented with bindings to a hexagon program written by the Air Force Wargaming Center.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Wargames, Ada, X Windows, Man Computer, Weather	143
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL